Unity-inspired object-oriented concurrent system development

by

Marlene Maria Ross

Thesis submitted in fulfillment of the requirements for the degree
Doctor of Philosophy in Computer Science
in the Faculty of Natural and Agricultural Sciences
University of Pretoria
Pretoria

January 2001

Unity-inspired object-oriented concurrent system development

by

Marlene Maria Ross

Thesis submitted in fulfillment of the requirements for the degree
Doctor of Philosophy in Computer Science
in the Faculty of Natural and Agricultural Sciences
Department of Computer Science
University of Pretoria
Pretoria

January 2001

Supervisor: Professor D.G. Kourie
Co-supervisor: Professor R.J. van den Heever

ABSTRACT

The design of correct software remains difficult, especially when dealing with concurrency. The primary goal of the research presented here is to devise a **pragmatic** software development method which

- aids the software designer in producing **reliable** software,
- is **scalable**,
- is **understandable**,
- follows a **unified approach** towards software development (is applicable to different implementation architectures),
- promotes **reuse**,
- has **seamless** transitions between the software development phases,
- guarantees **general availability** and **minimises developmental resources**.

The two main characteristics of the proposed new development method are captured in its name, viz. **Single Location Object-Oriented Programming (SLOOP)**. It is an **object-oriented** method, but its **computational model** is that of a set of statements that execute infinitely often and in any order. A program with such a computational model is called a Single Location Program (SLP). A UNITY program can also be classified as a Single Location Program. In the UNITY theory of programming it was demonstrated how this computational model could **simplify correctness reasoning**, particularly for concurrent systems.

It is this simplification, together with the structuring and reuse features of object-orientation, that is leveraged in the SLOOP method to produce a mechanism whereby **ordinary software practitioners** can take advantage of the benefits of a more rigorous approach towards software development without requiring an in-depth understanding of the underlying mathematics.

The following features of the SLOOP method contribute towards achieving the above goals:

- its **computational model** (it simplifies correctness reasoning, thereby promoting understandability and scalability, and also facilitates designs that are independent of the target implementation architectures),

- its **object-oriented** nature (apart from promoting reuse of frameworks, design patterns and classes, the SLOOP method provides the necessary mechanisms to facilitate reuse of correctness properties, correctness arguments as well as mappings to implementation architectures),
- its **emphasis on correctness reasoning** throughout the software development life cycle (its "constructive approach" aids reliability and seamlessness),
- the **unique** way in which the correctness properties can be specified, reused and reasoned about (this contributes towards understandability and scalability),
- the **checklist** of useful correctess properties that is provided (this promotes reliability),
- the **incorporation of existing notations** into the SLOOP syntax (this guarantees general availability, minimises developmental resources and aids understandability).

The main **contribution** of this thesis is that it presents a **unique** way of incorporating the SLP computational model into an object-oriented method with the specific aim of **simplifying informal correctness reasoning** and **promoting reuse**. The notation used for the specification of correctness properties facilitates reuse of correctness properties, ensures the integrity of these specifications and allows one to specify correctness properties at a higher level of abstraction.

The SLOOP method offers a **unique way of modelling concurrency in object-oriented systems** (via its parallel methods), which takes full advantage of the **encapsulation** and **inheritance** features of object-orientation. The issues surrounding **mappings** to implementation architectures are addressed, showing how even mappings can be **reused.** Finally, the **general applicability** of the SLOOP method is demonstrated.

SAMEVATTING

Die ontwerp van korrekte programmatuur bly moeilik, veral wanneer gelyktydigheid ter sprake is. Die primêre doel van hierdie navorsing is om 'n **pragmatiese** programmatuur ontwikkelingsmetode te ontwikkel wat

- die ontwerper sal help om **betroubare** programmatuur te ontwikkel,
- **skaaleerbaar** is,
- **verstaanbaar** is,
- 'n **eenvormige benadering** volg ten opsigte van programmatuurontwikkeling,
- **hergebruik** aanmoedig,
- 'n **gladde oorgang** tussen ontwikkelingsfases teweegbring,
- **algemene beskikbaarheid** waarborg en die **gebruik van ontwikkelingsbronne minimeer**.

Die twee hoofeienskappe van die voorgestelde nuwe ontwikkelingsmetode word weerspieël in die naam - **Enkel Posisie Objek-georiënteerde Programmering (EPOP)**. Dit is 'n **objek-georiënteerde** metode, maar die **verwerkingsmodel** is die van 'n groep stellings wat oneindig gereeld en in enige volgorde uitvoer. 'n Program met so 'n verwerkingsmodel word 'n Enkel Posisie Program (EPP) genoem. 'n UNITY program kan ook geklassifiseer word as 'n EPP. In die UNITY teorie van programmering is gedemonstreer hoe hierdie verwerkingsmodel die **beredenering van korrektheid** kan **vereenvoudig**, spesifiek vir gelyktydige stelsels.

Dit is hierdie vereenvoudiging, tesame met die strukturering en hergebruikseienskappe van objek-oriëntasie, wat aangewend word in die EPOP metode om 'n meganisme te produseer waar **gewone programontwerpers** die voordele van 'n meer formele benadering teenoor programmatuurontwikkeling kan benut, sonder om noodwendig die beginsels van die onderliggende wiskunde te hoef te bemeester.

Die volgende eienskappe van die EPOP metode dra by om bogenoemde doelstellings te bevredig:

- die **verwerkingsmodel** (dit vereenvoudig korrektheidsberedenering, dus verhoog dit ook verstaanbaarheid en skaleerbaarheid, en maak ook ontwerpe moontlik wat onafhanklik is van die implementasie argitektuur),
- die **objek-georiënteerde** aard (bo en behalwe die feit dat dit hergebruik van raamwerke, patrone en klasse aanmoedig, voorsien die EPOP metode die meganismes wat die hergebruik van korrektheidseienskappe, korrektheidsargumente, asook afbeeldings na die implementasie argitektuur moontlik maak),
- die deurgaande **klem op korrektheidsberedenering** tydens die progammatuur= ontwikkelingsfases (die 'konstruktiewe benadering' moedig betroubaarheid en 'n gladde oorgang tussen ontwikkelingsfases aan),
- die **unieke** manier waarop die korrektheidseienskappe gespesifiseer, hergebruik en beredeneer kan word (dit dra by tot verstaanbaarheid en skaleerbaarheid),
- die **lys van nuttige korrektheidseienskaptipes** wat verskaf word (dit moedig betroubaarheid aan),
- die **insluiting van bestaande notasies** in die EPOP sintaks (dit waarborg algemene beskikbaarheid, minimeer die gebruik van ontwikkelingsbronne en verhoog verstaanbaarheid).

Die **hoofbydrae** van hierdie verhandeling is die feit dat dit 'n **unieke** manier bied om die EPP verwerkingsmodel in 'n objek-georiënteerde metode te inkorporeer, met die spesifieke doel om **informele korrektheidsberedenering te vereenvoudig** en **hergebruik aan te moedig**. Die notasie wat gebruik word vir die spesifikasie van korrektheidseienskappe maak die hergebruik

van korrektheidseienskappe moontlik, waarborg die integriteit van hierdie spesifikasies en laat die spesifikasie van korrektheidseienskappe op 'n hoër vlak van abstraksie toe.

Die EPOP metode bied 'n **unieke manier om gelyktydigheid in objek-georiënteerde stelsels te modelleer** (by wyse van parallelle metodes), wat ten volle voordeel trek uit die **enkapsulasie** en **oorerwingseienskappe** van objek-oriëntering. Die kwessies rondom die **afbeeldings** op implementasie argitekture word geadresseer, en wys uit hoe selfs hierdie afbeeldings **hergebruik** kan word. Laastens word die **algemene toepaslikheid** van die EPOP metode gedemonstreer.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to the following people:

Professor D. Kourie, my supervisor, for his stimulating discussions, excellent advice and attention to detail. His encouragement and support have been invaluable.

Professor R. J. van den Heever, the co-supervisor, for his helpful suggestions, many stimulating discussions and careful reading of this thesis. His vision and insight regarding the various aspects of Computer Science have been a continued source of inspiration and motivation.

The management team at Azisa, who allowed me to work flexible hours in order to provide me with the opportunity to devote time to my studies.

My husband and parents, for their continued encouragement and all the opportunities that they have given me.

DECLARATION

I hereby declare that, unless otherwise indicated, all the work in this thesis was done by myself and that this thesis has not been submitted to this or any other institution of learning in support of an application for any other degree or qualification.

M. M. Ross
January 2001

DEDICATION

To Victor, for his love, support and understanding

and to Pieter Thomas and Emul, for being two such lovable boys.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The problem

Chandy and Misra state in their book on parallel program design: "The basic problem in programming is the management of complexity" [ChMi88]. Designing **correct** software, i.e. **software that complies with stated software requirements**, is a problem of considerable magnitude. The task becomes even more difficult when it involves concurrent and distributed systems. This is because of a multitude possible execution sequences resulting from interacting software artifacts.

A **concurrent** program comprises a number of sequential processes that execute simultaneously and may interact with one another [Bena90]. When these processes are not co-located, they form a **distributed** system. The processes in a distributed system communicate via message passing. If processes are co-located, they usually communicate via shared memory.

Whereas sequential programming is reasonably well understood, the challenges of concurrent and distributed programming remain daunting. As Ben-Ari observed: "Because of the possible interactions among the processes that comprise a concurrent program, it is exceedingly difficult to write a correct program for even the simplest problem" [Bena90].

The very nature of an object-oriented system, i.e. a collection of objects that interact with each other, suggests the notion of concurrency. However, initial object-oriented systems were focused on a single thread of control and since then a significant amount of research has been done to determine how best to implement concurrent objects. Active objects (used in Java) [Meye97, Lea96] and separate objects (used in Eiffel) [Meye97] are but two of the solutions that have been proposed. More detailed discussions of these approaches are given in Chapter 3.

Middleware support for distributed objects has also received a great deal of attention. Examples are the Common Object Request Broker Architecture (CORBA) project of the Object Management Group (OMG) [OHE97] and Microsoft's Distributed Component Object Model (DCOM) [MC-Web, CHYLS-Web]. The goal of a middleware product is to allow the software designer to work at a higher level of abstraction. For example, there is no need to concern oneself with name resolution or with the byte ordering used on a remote machine. Details such as how to serialise the data in order to transmit it to a remote processor are taken care of by the middleware product.

Classes are designed without taking their location into account. It should not matter whether they are local or at the other side of the world. The required CORBA services are simply multiply-inherited. If multiple inheritance is not possible, there are alternative ways of acquiring the relevant CORBA services, as will be shown in Chapter 8.

Although middleware infrastructures enable the software designer to focus on the system being designed rather than on the details of the supporting systems, it remains the responsibility of the

user of the middleware product to address issues such as **deadlock prevention** [FGHVE96]. The synchronous invocation of operations, i.e. where the client blocks until the method that it has invoked returns [Vino97], raises the possibility of deadlock. A scenario for deadlock under these circumstances is presented in Chapter 3. Various deadlock prevention strategies exist, as described in [Tane92], but the user of the middleware product still has to implement the appropriate strategy.

Interference, i.e. the many ways in which the interacting objects can affect each other due to multiple possible execution sequences, is another aspect that needs to be considered. The application designer has to ensure that the behaviour of the system is correct under all possible interleavings of the statements of the concurrent or distributed objects.

Compliance with the stated functional requirements of the system clearly remains the responsibility of the system designer. In order to be able to reason about the correctness of the system, aspects such as the semantics of operations should be well understood, i.e. it has to be clear what can be assumed about the effect of an operation immediately after it has been invoked [Meye97].

The semantics of an operation is determined by whether the interaction occurs synchronously or asynchronously. If an operation is invoked synchronously, the client has to wait until the operation has completed execution before it may continue. For the purposes of reasoning about the correctness of the system, the designer may assume that the postconditions of that operation hold when it returns. If an operation is invoked asynchronously, the client continues with its execution immediately after it has invoked the operation. (This is also called one-way invocation in CORBA terminology [Vino97].) In this case it cannot be assumed that the postconditions of the operation already hold at the time when the client continues its execution. In some cases a combination of the two types of invocation is also possible, e.g. the CORBA deferred synchronous invocation, where the client continues immediately after invoking an operation and obtains the result at a later stage [Vino97]. Whether an operation is invoked synchronously[1] or asynchronously clearly affects correctness reasoning.

Despite the existence of a plethora of techniques, tools and methodologies to design software for concurrent systems, ranging from formal to informal approaches, there is no general consensus regarding best approaches, however well defined. Most specification languages have carefully chosen features that make them more appropriate for certain systems than others [Mori90]. Many formal techniques suffer from **scalability** and **understandability** problems [Meye90, RPS95, Sifa99], while it is often difficult to reason about the correctness of systems produced using informal methods. In [GrSc99] it is argued that by improving the teaching methods of the underlying mathematics, formal proofs will no longer be so daunting. However, since most practising software designers still lack this type of training, alternative options are worth investigating.

In summary, when designing systems of concurrently executing and possibly communicating artifacts, the following problems need to be addressed:

- Although middleware infrastructures enable the designer to work at a higher level of abstraction, the fundamental software correctness properties still need to be considered. The onus is still on the designer to ensure that the system behaves as expected. Not only must the

[1] The term synchronous is used differently in synchronous languages such as Esterel, where it means that once an input event is processed, the output resulting from the occurrence of the input event is computed without any delay, i.e. the output is strictly synchronous with the input [BeBe97]. The producer of the input event **does not wait for a result**. An input event can be broadcast to multiple components that all react simultaneously to the same input event.

functional requirements be met, but issues such as deadlock prevention and interference have to be addressed. Thus, the **reliability** of the system must be ensured.

- The second issue is **scalability**. As the size of the system increases, the more difficult it becomes to reason about the correctness of the system.
- It is necessary to be able to understand both the design and the correctness arguments associated with the design with relative ease, i.e. **understandability** is important.

1.2 The goals

The **primary goal** of the research represented in this thesis is to devise a software development method[2] which addresses the problems listed in the previous section. Thus, it has to promote **reliability** in the software systems produced via this method, and it has to be **scalable** as well as **understandable**. However, it is prudent to consider what other features are also desirable. Consequently, the following **additional goals** have been identified.

The software development method should be suitable for **all types of systems**, i.e. sequential, concurrent and distributed. A **unified approach** should be followed for the design of the system; only during the implementation phase should a mapping to a specific target architecture be considered. This goal also implies that the method should support both **synchrony** and **asynchrony** as fundamental concepts [ChMi88], since there are certain types of systems that are inherently synchronous (e.g. systolic arrays), while others have asynchronous behaviour (such as telecommunications systems comprising multiple nodes). It should also be possible to model **non-determinism**, since some systems are inherently non-deterministic (e.g. operating systems) [ChMi88].

Reusability is widely recognised as being of paramount importance in maximising efficiency of software production. This notion is extended here to apply not only to class and design reuse, but also to the reuse of correctness arguments and mappings to target architectures.

An important goal is **seamlessness**, i.e. using the same development process throughout the entire software lifecycle [Meye97]. Sifakis [Sifa99] lists the distance between formal languages (used for specification) and the programming languages (used for implementation) as one of the obstacles to a more widespread use of formal methods. The study of executable languages for system design and modelling is therefore becoming one of the most important research directions in formal methods [Sifa99]. Apart from the advantage of using the same concepts during all phases of software development, seamlessness has another benefit: **rapid prototyping** can be facilitated if the design notation is executable or can be mapped fairly easily to an executable language.

The last objective is that the supporting infrastructure for the method should, as far as possible and feasible, be based on existing notations and development environments in order to **guarantee general availability** and also to **minimise developmental resources** for the method. As stated in [Mori90], if a notation is based on "familiar, simple concepts" and has "powerful yet simple combining forms", it would promote its widespread usage.

1.3 Towards a new method

Aspects from various existing approaches are synthesised in a unique way in order to arrive at a method which is relatively easy to use and understand, while reusability and the ability to reason

[2] The term "method" is considered more appropriate than "methodology" in this context. In [Bjor99] "method" is defined as "a set of principles for selecting and applying techniques and tools in order efficiently to analyse and synthesise (i.e. construct) efficient artifacts (here: software)". "Methodology" is defined as the "study and knowledge of methods".

about the correctness properties of a program are integral aspects thereof. The method capitalises on the synergy of the following:

- the **object-oriented** paradigm,
- the concept of programs without any locus of control, as applied in **UNITY**,
- **formal methods** and
- **reflective computation**.

The above features are now discussed in more detail.

1.3.1 The object-oriented paradigm

In the quest for higher quality software and greater productivity, much has been achieved since the publication of Dijkstra's 1976 "A discipline of programming" [Dijk76]. While structured analysis and design methodologies are still widely used, the object-oriented paradigm is already well entrenched.

All aspects from the object-oriented paradigm are incorporated into the new method. This includes **encapsulation**[3], **polymorphism**[4], **inheritance**[5] and more recent developments such as **design patterns** and **frameworks**. A design pattern describes a **recurring problem** together with the **core solution** to that problem. The latter is given in such a way that it may be used repeatedly without necessarily resulting in duplicate implementations [GHJV95]. A framework can be defined as "a set of classes that embodies an abstract design for solutions to a family of related problems and supports reuse at a larger granularity than routines or classes" [BGL93, JoFo88]. The framework user customizes the framework via **subclassing** and by **creating instances** of its classes [GHJV95]. A framework often combines a number of design patterns in its solution.

The new method takes full advantage of all the excellent **structuring** and **reuse** capabilities of object-orientation.

1.3.2 UNITY

UNITY (Unbounded Nondeterministic Iterative Transformations) is a **computational model** and **proof system** devised by Chandy and Misra [ChMi88]. Their work is a departure from the conventional model of control flow in the sense that they maintain that the notion of a program location pointer is superfluous. Any program can be written in terms of a set of assignment statements that are all executed infinitely often. There are therefore none of the usual programming language constructs, such as `if then else` statements, `for` loops, etc. Instead, each statement is a multiple assignment statement which may be conditional and which is executed infinitely often. Each multiple assignment statement executes **atomically**.

Gerth and Pnueli [GePn89] refer to this class of programs as **Single Location Programs** (SLPs). They argue that fewer syntactic structures make it easier to reason about a program. However,

---

[3] Encapsulation (also information hiding) is the capability of "separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from the other objects" [RBPEL91]. Rumbaugh et al. state further that "encapsulation is not unique to object-oriented languages, but the ability to combine data structure and behaviour in a single entity makes encapsulation cleaner and more powerful than in conventional languages that separate data and behaviour" [RBPEL91].

[4] Polymorphism is "the ability for an entity to become attached to objects of various possible types" [Meye97]. This should be controlled by inheritance.

[5] Inheritance is the mechanism whereby a class incorporates features from another class in addition to its own [Meye97].

universality should be maintained, i.e. it should still be possible to compute any computable function using this restricted class of programs. Their formal definition of SLPs is as follows: "The class SLP - single location programs - is a universal class. Here, an SLP-program has the form I: *[A_1 [] ... [] A_n]; A_1, ..., A_n are conditional assignments and I specifies the initial state."

Thus, the state I holds initially. The "*" symbol indicates that the section in square brackets is executed infinitely often. The "[]" symbols separate the conditional assignments and indicate that any one of these conditional assignments may be executed during each iteration. UNITY also has a **fairness** requirement, viz. each statement must be executed infinitely often.

This approach greatly **simplifies reasoning about the correctness** of the program, since the emphasis is on the properties of the program as a whole as opposed to proving properties related to control flow. The major deficiency of specification formalisms that rely on control flow in parallel programs is the "complexity of reasoning about computation histories" [GePn89].

Gerth and Pnueli state: "... it took a rare insight to see that the results [of simplifying programs into SLPs] would be more than a mildly interesting theoretical result, because many concurrent algorithms can be developed in such an impoverished language. Chandy and Misra had that insight." [GePn89].

The UNITY approach towards program design is to **ignore the target architecture during the initial stages**. Once it has been shown that the design is correct, the UNITY program is mapped to the required target architecture. This involves the association of variables and statements to various processes and processors. More detail regarding the UNITY computational model and proof system is given in Chapter 2. At this stage it suffices to say that the above features of UNITY are the most important ones that have been incorporated into the new method proposed in this thesis.

1.3.3 Formal methods

Formal methods are "the set of activities - specification, reasoning, refinement - that add mathematical rigour to the development, analysis and operation of computer-based systems" [RPS95]. The sound mathematical basis of a formal method is typically provided by a formal specification language [Wing90].

A myriad of formal methods exist, for example, Z [Spiv92], the Vienna Development Method (VDM) [Jone86], B [Abri96], Communicating Sequential Processes (CSP) [Hoar85], Larch [GuHo93] and Temporal Logic of Actions (TLA) [Lamp94]. Most of these methods have specific applications domains [Wing90]. For example, VDM is a development method that is well-suited to sequential systems, while CSP is appropriate for the specification of concurrent and distributed systems. Object-oriented extensions have been developed for many of these formal methods [RPS95]. Some of the methods that were originally developed for sequential systems have been extended to cater for concurrency as well [Jone99].

The standardization bodies have also been active in the area of formal methods. Formal Description Techniques such as LOTOS [ISO89] and ESTELLE [ISO97] have emerged from the International Standards Organisation (ISO). The International Telecommunications Union (ITU) produced the Specification and Description Language (SDL) and in 1993, the ITU Z.100 publication [ITU-T93] incorporated object-oriented concepts into SDL.

Due to the complexity of concurrent and distributed systems, the need for formal methods has become more urgent. Two broad classes of formal methods can be identified, viz. synthesis and verification methods [Jone99]. A synthesis method constructs a program from a specification [Jone99]. **Verification** is the process of showing that a program satisfies its specification

[Wing90]. Francez [Fran92] defines a **specification** of a program as "a collection of criteria, which, if met by that program, would qualify that program as correct (with respect to those criteria)".

In order to ensure that the specification is **unambiguous**, the specification language has to have precise semantics. That means that every statement in the language should have exactly one meaning [Jone80]. A specification should also be **complete** (i.e. all the important properties should be specified [JiZh96]) and it should be **consistent** (i.e. it should not contain any contradictory requirements [JiZh96]).

The **advantages** of applying a **formal method** during software development are:

- It **reveals ambiguity, incompleteness and inconsistency** in a system [Wing90].
- It **promotes** a **systematic**, rather than ad hoc, approach towards **specification, development and verification** [Wing90]
- Ultimately it **facilitates automatic verification** of the software [RPS95].

An interesting side effect of program verification is that it has highlighted the merits of more formal methodologies, since it has illustrated that well structured programs are easier to verify [Fran92].

Despite the obvious advantages of formal methods, this field is still regarded by many as rather **esoteric** [Seli93, Sifa99], mainly because the underlying mathematics is perceived as **difficult** and **tedious** to use. One of the aims of the new method presented here is to **encourage** a more rigorous approach towards software design without requiring the users of the method to have an in-depth knowledge of formal methods. Thus, it needs to be **usable by practising software designers** (who are not applied mathematicians) in real-life projects.

The method should be based on a sufficient amount of formality and rigour in order to **support reasoning about correctness**, preferably in an informal **lightweight** style. The term "lightweight" as used here refers to the judicious use of mathematical techniques during system design. Thus, formal proofs are avoided and the specification is only formalised to the extent that reasoning about correctness is facilitated. This is similar to the "formal methods light" approach proposed by Jones in [Jone96]. He argues that it is "important to understand the formal basis but to use - in most cases - a less than completely formal approach" [Jone96]. Many errors can be detected by using informal arguments and the benefits of complete formalisation often do not justify the cost.

As noted by Gries [Grie96]: "More liberally, any informal use of theoretical ideas in the development process can be viewed as an application of formal methods. Examples are the use of mathematical notation for part of a specification, the use of an informal invariant and bound function when developing a loop, and the use of an informal coupling invariant that describes how an abstract data type is to be implemented".

Not only is it important to specify the functionality of a system and subsequently produce a program to meet the requirements of the specification, but it is also important to **document why the designer believes the program to be correct** [Jone80]. The new method promotes this style of software development. It addresses the goals described by the **first two** bullets listed above. A formal verification method with its associated formal semantics and proof system is not proposed. A **pragmatic** approach towards the application of formal methods is therefore followed.

The new method exploits the **combination** of the notion of formal methods with the structuring capabilities of object orientation. As a result the feasibility of adopting a more formal approach towards the design of **medium to large systems** is increased. The specific example chosen to

elucidate various aspects of the method illustrates the **applicability** of the method to **non-trivial** problems.

1.3.4 Computational reflection

Computational reflection can be defined as "the behaviour exhibited by a reflective system, where a reflective system is a computational system which is about itself in a causally connected way" [Maes87]. Thus, the system is made **self-aware**. In [BMRSS96] a reflection architectural pattern is described. The architecture is split into two levels: the **base level** containing the **application logic** and the **meta level** containing a **self-representation**. The meta level contains information about the structure and behaviour of the system.

Several useful **applications of computational reflection** are listed in [Bekk93]. The ones that are particularly relevant to this research are:

- **Debugging**: Trace statements should not be part of the application logic. By making them part of the meta-object, the application logic remains unchanged, whether the trace statements are executed or not.
- **Assertion checking**: Messages are intercepted by the meta-object in order to evaluate pre- and postconditions.
- **Reasoning about control**: The meta-object may be used to calculate which statement to execute next.
- **New paradigms**: New concepts may be experimented with by first implementing them in the meta-objects.

1.3.5 The resulting method: SLOOP

The new method is called **SLOOP (Single Location Object-Oriented Programming)**. The name reflects the marrying of the UNITY computational model and object-oriented concepts.

The SLOOP method encompasses the **analysis, design and implementation phases** of system development. Throughout the development lifecycle the emphasis is on the correctness properties of the system. Instead of performing *a posteriori* program verification, correctness reasoning forms an integral part of the program construction process. This is called the **"constructive approach"** to software correctness [Meye90]. Examples of the various types of correctness properties that can be considered are discussed in detail in Chapter 5.

A SLOOP specification can be viewed as a **"mixed specification"** [Sand90], i.e. the specification may contain **properties** specified via programming logic as well as **SLOOP statements** that can be mapped to an executable program. The correctness properties describe the requirements which need to be complied with by the various software artifacts in the system, while the SLOOP statements provide a design specification.

First, the required functionality of the system is specified using the programming logic described in Chapter 4. SLOOP statements are then derived and informal correctness arguments are presented to describe why the statements satisfy the specified correctness properties. The programming logic used in the SLOOP method is based on the UNITY programming logic defined in [ChMi88]. Finally, the SLOOP statements are mapped to an executable program, taking care that the correctness properties are preserved. **Several refinements** may be required.

SLOOP statements are similar to UNITY multiple assignment statements, but in addition to the latter, the syntax allows the statements to contain Smalltalk message expressions.

The validity of this approach is confirmed by the views of Gerth and Pnueli [GePn89]: "A question with theoretical as well as practical significance is how far the SLP-syntax can be extended again while maintaining the simplicity of the proof system. One possibility is given by noticing that the particular form of the actions within the iteration of an SLP-program does not matter. It is the fact that these actions are executed **atomically** that counts. Accordingly, we can allow arbitrary programs instead of assignments as the atomic actions. The proof system would change only by the addition of rules to reason about these atomic programs; the existing rules would not change."

The SLOOP method is an object-oriented method, but it is **not** based on the traditional computational model. A system consists of a number of objects. Two types of operations[6] are defined for SLOOP objects, viz. **sequential** and **parallel**. The purpose of a sequential operation is similar to that of a **terminating function**[7] that may be called from within a UNITY statement. The statements of a sequential operation are executed sequentially (in the order of their appearance) and each statement is executed at each invocation of the operation.

The nature of the computational model used in SLOOP is evident from the characteristics of the parallel operations. Each parallel operation contains one or more parallel statements, i.e. statements that could be interleaved in any arbitrary order. Each parallel statement should be executed infinitely often (this is the fairness requirement). Only one parallel statement is executed at each invocation of a parallel operation[8]. The parallel operations that are selected[9] for a program are **invoked** infinitely often. The **core** of a SLOOP program is this **set of parallel statements**. They may be executed in any order, provided the **fairness** requirement is satisfied.

Note that the concept of a sequential operation is not essential in the SLOOP method. As stated earlier, the SLP class of programs is a universal class, i.e. all computable functions can be computed via Single Location Programs. This implies that any computation can be written in terms of a set of SLOOP parallel statements. As a result, any computation represented by a sequential operation can also be written in terms of a set of parallel statements. The notion of a sequential operation can be viewed as syntactic sugaring for the convenience of the software developer. One advantage of the sequential operation construct is that it makes it possible to utilise existing class libraries. Another benefit is that it allows one to write operations in terms of the conventional execution model when

- the computation is so simple that writing it in terms of a set of parallel statements would not simplify correctness reasoning and
- when it is unlikely that the operation will be implemented via parallel processing.

The rationale for including sequential methods in the SLOOP method will be discussed in more detail in Chapter 4, Section 4.2.3.

The fact that the execution order of the parallel statements is unspecified allows one to model non-determinism. The ability to represent non-determinism is one of the advantages of UNITY over functional programming [ChMi88] and is therefore one of the reasons why the SLOOP method is based on UNITY rather than on functional programming. The concepts of sequential and parallel operations and statements are described in detail in Chapter 4.

---

[6] The Smalltalk philosophy of viewing an object as consisting of some private memory and a set of operations [GoRo89] is adopted in the SLOOP method. Smalltalk operations are similar to the sequential operations defined for SLOOP.

[7] Examples of terminating functions used in UNITY [ChMi88] are: min (it determines the minimum of two values), max (it determines the maximum of two values), odd (it returns true if the argument is odd), even (it returns true if the argument is even).

[8] The rationale for this characteristic of the SLOOP method is given in Chapter 4.

[9] Multiple parallel operations may be defined for a class. Only those that apply to the application under development should be activated, i.e. should be part of the list of operations that are invoked infinitely often.

Programs are designed in a **unified** manner: the initial solution does not target a specific architecture, such as sequential or concurrent. The resulting SLOOP statements are then mapped to one or more processes according to specified rules. More parallelism may be introduced through a **series of refinements**. During each refinement the existing correctness properties have to be preserved and new ones may be added.

Even though the UNITY method simplifies reasoning about concurrent systems, it is still a cumbersome process when medium to large systems are involved. Since the concept of **reusability** was first promoted, it has progressed from code and class reuse to design reuse. Reasoning about the properties of a system is another candidate for reuse and is therefore incorporated into the SLOOP method. Thus, the method is used to specify and reason about the behaviour of and collaboration amongst the classes constituting a system. **The correctness arguments become part of the reusable building blocks**. The latter can be reasoned about individually and in combination with others.

The SLOOP method advocates a combination of the top-down and bottom-up software development approaches. As stated in [Hoar99], the top-down and bottom-up approaches are complementary in any scientific or engineering discipline and can be mixed or rapidly alternated. In the SLOOP method, the initial analysis of the system is performed in a top-down manner, but thereafter the emphasis is on reusing existing building blocks. However, when an appropriate reusable artifact cannot be found, the top-down and bottom-up approaches are once again alternated, this time at the level of the new artifact being developed.

In order to gain the full benefit from such an approach, it is important that the correctness properties should form part of the reusable building blocks. This echoes the views of Francez: "Recently, the bottom-up approach has been getting the attention it deserves. This approach calls for the construction of building blocks that have been verified, and provides rules for their correct combination in various ways to obtain larger designs" [Fran92].

The SLOOP notation is based on UNITY and **Smalltalk** [GoRo89]. Incorporating an existing object-oriented language into the SLOOP method has several advantages:

- it aids **understandability**,
- it facilitates a **relatively easy transformation** of the specification into an **executable program** as illustrated in Chapter 8 and
- an **extensive class library** is immediately accessible.

The rationale for selecting Smalltalk as the basis for representing object-oriented concepts is

- its **suitability for experimental systems**,
- its **reflective facilities**,
- the **simplicity** and **elegance** of its design, and
- its **automatic garbage collection** facilities.

In Smalltalk, objects are treated as "first class citizens" in a unified, syntactically simple fashion. In contrast, for example, Java has a "confusing modular structure with three interacting concepts (classes, nested packages, source files)" [Meye97], while the rather complicated C++ can be viewed as a transition technology: it enabled those familiar with C to make to the transition to object-oriented technology [Meye97]. Although SLOOP programs contain Smalltalk message expressions and are mapped to executable Smalltalk programs in the examples in this thesis, mappings to other object-oriented languages such as Java are also possible. Smalltalk to Java translation has already been investigated by several researchers [Enko98]. One would just have to be aware of the weaknesses of Java [ABV00] and take the necessary steps to deal with them.

As will be shown in Chapter 8, the reflective facilities of Smalltalk are harnessed during the implementation phase to ensure that meta level information (such as which statement to select for execution) is kept separate from the base level information (the application logic).

The SLOOP **notation** is **formally specified** in Backus-Naur Form (BNF). Its **semantics** is defined **informally** in natural language. In principle it is therefore possible to build various tools, such as a SLOOP syntax checker and a SLOOP-to-Smalltalk translator, or even a SLOOP compiler. However, that is beyond the scope of this work. The **purpose** of this research was primarily to determine how well an object-oriented software development method based on a different computational model could address the issues listed in Section 1.1. The results are very encouraging, as will be reported below.

Several example SLOOP programs have been developed by the author. Some of them were non-trivial, as illustrated by the one presented in Appendix B. The most remarkable aspect of these experiments is the fact that **no modifications to the design or the logic were required once the design phase had been completed**. The executable programs produced the correct results when they were first run. It is strongly believed that this can be attributed to the **"constructive approach"** design philosophy. The emphasis on correctness properties during the analysis, design and implementation phases promotes a **disciplined** and **careful** approach towards the software development process. A high quality design was achieved without going to the lengths of applying a completely formal method.

Another important result is the impact of the **computational model**. Not only does it **simplify correctness reasoning**, but it is also possible to map a SLOOP program first to a **sequential** executable program and then to a **concurrent** one **without changing a single SLOOP statement**. Both mappings produced the correct results when they were first run.

In summary, the **goals** stated earlier are met in the following way:

- The "constructive approach" promoted by the SLOOP method encourages the designer to work in a disciplined way, paying attention to correctness properties at all stages of the system development. This is conducive to a more **reliable** end product.
- The expressive power and the atomicity of the SLOOP parallel statement facilitate a design at a high level of abstraction[10]. All the actions that should be performed without interference are grouped into a single parallel statement. This enables one to take care of issues such as deadlock prevention and mutual exclusion in a very simple way during the design phase. When the SLOOP program is mapped to a specific target architecture during the implementation phase, the handling of deadlock prevention and mutual exclusion remains simple, provided the atomicity of each parallel statement is preserved during the mapping. Possible ways of ensuring the atomicity of the parallel statements during the implementation phase are described in detail in Chapter 8.
- The object-oriented nature of the method addresses the goal of **scalability**.
- The computational model simplifies reasoning about correctness properties, which addresses the issue of **understandability**. It is not necessary to take location counters into account. By definition, the order in which the parallel statements of a SLOOP program are executed, is irrelevant. The statements therefore have to designed in such a way that the program will be correct regardless of the execution order of the statements. Correctness properties are universally or existentially quantified over all program statements.
- Reasoning about correctness properties comprises showing informally that the SLOOP program statements indeed satisfy the stated correctness properties. Familiarity with the appropriate mathematical theorems is not required. This promotes its use by software designers even if they have not been trained in the application of formal methods.

[10] Abstraction is defined as "the selective examination of certain aspects of a problem" [RBPEL91]. The goal of abstraction is to highlight those aspects that are relevant to the issue currently being dealt with and to suppress other aspects.

- The programming logic used to specify the correctness properties sufficiently supports informal reasoning about them.
- The programming logic is based on the familiar concepts of pre- and postconditions and the SLOOP statements are based on Smalltalk-80, a well-known object-oriented language, which reduces the learning curve.
- The design approach is based on the UNITY concept of first developing a general solution based on the required properties, which is then refined for specific architectures. This satisfies the goal of a **unified** approach.
- The object-oriented nature of the method facilitates class and design **reuse**. In addition, correctness properties and correctness arguments associated with a class and its methods are also reused. Mappings to various target architectures represent yet another form of reuse.
- The same notation is used during analysis and design, and if Smalltalk-80 is chosen as the implementation language, there is a **seamless** transition between the various development phases.
- The fact that a SLOOP program can be mapped to an executable Smalltalk-80 program fairly easily, satisfies the goals of **general availability** and the **minimisation of developmental resources**. It also allows for **rapid prototyping**.

When devising a new software development method, the above issues are only a subset of all the aspects that need to be considered. The smaller (but also important) issues are covered in the detailed discussions of the SLOOP method in the remaining chapters of this thesis. For example, the applicability of the method to various **types** of design problems is addressed in Chapter 9.

1.4 Related work

Over the years many software development methods have been devised, ranging from informal through semi-formal to formal. It is still a fertile research area as is evident from the literature. In the realm of formal methods, there is ongoing research in all of the methods listed in Section 1.3.3. For example, the support for concurrency in VDM is an area that has already been addressed, but still needs more investigation [Jone99]. The combination of different formal methods, such as CSP and B, has also been considered [Butl99].

Another area that has attracted a lot of interest is the development of formal methods for object-oriented systems [TMP99]. This includes object-oriented distributed system specifications. For example, in [Sivi97] a formal method based on temporal logic is proposed which enables one to specify the correctness properties of distributed system components as well as reason about those properties. It assumes that all operations are invoked asynchronously and it also has the restriction that specification statements can refer only to properties that are local to a single component.

As far as semi-formal methods are concerned, the Unified Modelling Language (UML) [RSC-Web] has emerged as the notation of choice for many software developers. Many researchers are now investigating the formalisation of the notation. UML has been augmented with the Object Constraint Language (OCL) in order to facilitate the definition of integrity constraints as part of the class diagrams (previously they could only be specified as informal textual annotations) [MaCe99]. Another possibility that is being investigated is the transformation of semi-formal specifications into formal ones. One example is the transformation of OMT specifications into B specifications [MeSo99].

The SLOOP method is a semi-formal method which has as its basis a formal notation for specifiying the structure and behaviour of a software system. It is intended for software practitioners rather than theoreticians and is therefore not aimed at competing with formal methods such as VDM, Z, B and CSP (and their many variants). Research within the UNITY

framework has also focused on more formal aspects [ChCh99]. The SLOOP method will instead be compared with other methods that address roughly the same problem space as the SLOOP method.

The most significant difference between SLOOP and most other semi-formal methods is its computational model. Since the SLOOP computational model greatly simplifies correctness reasoning, the SLOOP method has a distinct advantage over methods such as UML [RSC-Web] and the Business Object Notation (BON) [PaOs99][11] that use conventional computational models.

Action systems have a mathematical model that is equivalent to that of UNITY [BaKu89] and therefore also to that of SLOOP. Recent work on action systems include some extensions to support modularisation [BaSe94] and the addition of object-oriented constructs [Kurk96]. The concepts described in [Kurk96] have been implemented in an experimental language called DisCo (*Dis*tributed *Co*operation). Although DisCo is categorized as a formal specification language [RPS95], it is stated in [Kata-Web] that the notation is appropriate for the complete spectrum of users, from software practitioners to theoreticians. The SLOOP method differs in several ways from the work presented in [BaSe94] and [Kurk96], most notably the following:

- The extension to action systems as described in [BaSe94] does not deal with object-orientation. It shows how Dijkstra's guarded command language [Dijk76] can be extended with the concept of procedures (including local, imported and exported variables), i.e. it adds support for modularisation. It also describes how the language Oberon can be modified to support action systems. Oberon is particularly suitable because it already supports concepts such as imported and exported variables, etc.
- Modular action systems [BaSe94] do not have object-orientation features such as polymorphism and inheritance.
- Multiple partitioning options are available for the actions and variables of modular action systems. This raises the possibility of problems such as deadlock. In the case of modular action systems the solutions to the problems associated with mappings are not treated as reusable artifacts. In contrast, this is an important aspect of the SLOOP method.
- In the object-oriented action system described in [Kurk96] the actions and classes are viewed as **separate entities**. Multiple classes may participate in a joint action. The actions **replace** the concept of methods. A class therefore has attributes, but **no methods** associated with it. An action **does not form part** of a class. The statements in an action may **modify the variables** of the objects participating in the action directly.
- SLOOP parallel methods form an **integral** part of a specific class. It may invoke methods of other class(es) and in that sense there is synchronisation with the other class(es), but **a parallel method belongs to a single class** and can only be refined by the subclasses of that class. SLOOP parallel methods are therefore in line with the concept of **encapsulation** and information hiding. The contents of the parallel method is not important to any class other than the containing class and its subclasses. Only its correctness properties need to be visible. A SLOOP parallel statement may invoke sequential methods, which ensures that the **structuring** capabilities of object-orientation are exploited fully.
- Another difference between object-oriented action systems and SLOOP is the way in which **inheritance** is handled. In the case of object-oriented action systems the preconditions of an operation can be strengthened during specialisation. This is the opposite of the inheritance rule in SLOOP, which specifies that preconditions may not be strengthened (they may remain the same or be weakened). This is to ensure that subclasses will always accept requests from clients that are unaware of the fact that they are not dealing with the parent class (useful in polymorphism) [Meye97]. The rationale for this inheritance rule in SLOOP is discussed in more detail in Chapter 4.

[11] BON is designed to work seamlessly with Eiffel [Meye97], i.e. a BON specification is transformed into an Eiffel program during the implementation phase.

- Although the computational models of SLOOP and actions systems are similar, there is a subtle difference. The **fairness requirement** of the SLOOP computational model specifies that each parallel statement has to execute infinitely often. This obviates the need for guards, resulting in the simplification of the correctness arguments. In the case of action systems, a similar fairness requirement does not exist. As a result, a guard has to be associated with each action. Only those actions of which the guards are enabled, may be selected for execution. This is to ensure that a statement which is not enabled is not selected for ever, preventing progress. Fairness requirements must be specified explicitly for individual actions.
- In the SLOOP method the emphasis is on specifying a **set of correctness properties** and using those as the basis for the **derivation** of the SLOOP program. Informal correctness reasoning consists of providing correctness arguments showing that the statements of the SLOOP program satisfy the specified properties. The SLOOP class and method specifications **include correctness property specifications**. In the case of DisCo (which is based on object-oriented action systems), the emphasis is on specifying the system in terms of a set of classes and actions. Class specifications may contain invariants. The actions are specified in terms of the modifications to the relevant variables. An animation tool is provided to validate the DisCo specification against the **informal requirements** of the system. Verification of **invariant properties** can be performed using a prototype tool which employs a Prototype Verification System (PVS) theorem prover.
- The **formal basis** for actions systems is TLA [Kurk96], whereas that of SLOOP is based on the UNITY programming logic [ChMi88].
- The issues involved regarding the **mapping to various architectures** are not considered in [Kurk96]. SLOOP explores this aspect to a great level of detail.
- **Seamlessness** is a very important aspect of the SLOOP method. The DisCo method focuses on the analysis and design phases of the software development life cycle.
- The emphasis in [BaSe96] and [Kurk96] is on formally applying **refinement calculus** on actions systems that are extended to include modularisation or object-oriented concepts respectively. The focus in SLOOP is to take advantage of the **simplifying** aspects of the computational model in order to enable informal reasoning about correctness of object-oriented systems. The SLOOP method therefore provides a list of correctness properties and shows why they are important for design when an informal approach is used.

1.5 The contribution

It is evident from the above that there have been many approaches towards managing software complexity. The SLOOP method proposed in this thesis achieves this by combining the **simplicity** resulting from the UNITY computational model with the **structuring capabilities** of the object-oriented approach. It offers all the benefits of a **true object-oriented method**, such as encapsulation, polymorphism and inheritance, and at the same time it provides an elegant model to describe **concurrent** behaviour.

It equips **practising software designers** with a software development method which offers many of the advantages of a more formal approach, while the underlying mathematics need not be considered. The method **guides** the software designer to **focus on correctness properties** during the analysis, design and implementation phases of the software development life cycle, i.e. the software designer is aided in following a "constructive approach" towards software development.

The value of the method lies in its simplification of the correctness arguments pertaining to the behaviour of interacting objects. The **understandability** and **reusability** features of the method make it **feasible** to reason about the correctness of medium- to large-scale systems in an informal way. The **correctness properties** form part of the reusable building blocks.

The SLOOP method also makes it possible to **reuse mappings** to various implementation architectures. This is due to the fact that the issues related to the target architectures do not form part of the design considerations. **Generic** solutions for mapping SLOOP statements to executable programs are proposed, which can be reused by any application during the implementation phase.

The results of specifications using this formalism have been promising. It has been especially encouraging to note how effectively building blocks could be reused in new systems. The **general applicability** of the SLOOP method to various types of design problems has also been confirmed by applying it to various design patterns.

The significance of the UNITY computational model with respect to the simplification of correctness reasoning was pointed out in [ChMi88]. The SLOOP method has been developed to build on that concept, but with a different audience in mind: the ordinary practising software designer.

1.6 Structure of the thesis

The structure of the thesis is now presented. An overview of the layout is given first and subsequently more detail is presented regarding the contents of each chapter.

To start with, some background is given regarding the theories and principles underpinning the SLOOP method. Next, the syntax and semantics of the SLOOP method are presented, as well as an overview of its application during the various software development phases. In the subsequent chapters a case study is used to elucidate various aspects of the application of the SLOOP method during the analysis, design and implementation phases.

The relevant parts of the case study are presented in the appropriate chapters, while the complete SLOOP specification is given in an appendix. The case study is **non-trivial**. There are several reasons for choosing this style of presentation:

- It illustrates that the method can be used to develop **non-trivial** systems.
- It provides a vehicle for demonstrating the **wide range of correctness properties** that need to be considered during system development.
- The example is sufficiently complex that it can be used to **elucidate all the aspects of system development** that are addressed in this thesis.
- It provides a **single coherent picture** of what is involved during system development when the SLOOP method is being used.
- The problem statement can be given only once; **many examples build on what has gone before**. If multiple toy examples had been used, it would have been necessary to provide a problem statement for each one, as well as the required background that leads up to what is being demonstrated.

The remainder of the thesis is structured as follows:

Chapter 2 contains a brief discussion of some of the issues related to the specification of correctness properties, culminating in a detailed description of UNITY, the method proposed by Chandy and Misra. Chapter 3 also forms part of the literature survey, covering the reusability aspects of software design. Some of the approaches towards handling concurrency in the object-oriented paradigm are discussed. Frameworks, design patterns and various models for system design are investigated.

Chapter 4 introduces the SLOOP method. The syntax and semantics are given here. Examples are used to illustrate the basic principles. An overview of the application of the method in system development is presented. In Chapter 5 it is shown how the new method is used during the analysis phase. This is followed by a description of its use during system design in Chapter 6.

Chapter 7 discusses the various issues surrounding correctness reasoning. Chapter 8 covers the heuristics for the mapping of SLOOP programs to various architectures. Chapter 9 discusses how design patterns can be incorporated into SLOOP designs. It demonstrates the applicability of the method to various types of design problems.

The conclusions are presented in Chapter 10. It evaluates the SLOOP method in terms of the goals mentioned in Section 1.2 and offers some final remarks.

Appendix A contains a quick reference to the SLOOP notation and Appendix B contains the complete SLOOP specification of the case study used in the body of the thesis.

CHAPTER 2

CORRECTNESS, SPECIFICATIONS AND UNITY

2.1 Introduction

The characteristics of the SLOOP method can be divided into two broad categories, viz. **software correctness** and **object-orientation**. This chapter serves as an introduction to the issues related to the software correctness aspects of the method. Similarly, Chapter 3 covers object-oriented aspects. Together these two chapters provide the **background** to the concepts on which the SLOOP method is based.

Software correctness can be defined as the **compliance** of the software with **specified** correctness properties. In Chapter 1 it was stated that, despite the emergence of middleware infrastructures such as CORBA, it is still the responsibility of the software designer to ensure that the necessary **software correctness properties** are satisfied. This chapter elaborates on exactly what this comprises.

First of all, arguments are presented to justify the **"constructive approach"** [Meye90] towards software development. It is shown why it is not sufficient to rely **only** on **testing** to ensure that a reliable and functionally correct product is produced. The reasons for preferring the "constructive approach" to *a posteriori* verification are also given.

When dealing with **concurrent** systems, correctness reasoning is simplified if the concurrency is modelled by the **arbitrary interleaving** of the statements of the constituent components of the program [MaPn81a]. The justification for this abstraction is given, followed by a discussion of related issues, viz. **interference**, **atomicity** and **fairness**. The application of these concepts to the SLOOP method is discussed in Chapter 4.

As will be demonstrated in Chapter 5, the SLOOP method relies heavily on the analysis of the problem statement in terms of a set of correctness properties. The question therefore arises: which correctness properties should be specified? Manna and Pnueli define **a useful set of correctness properties** in terms of **temporal logic** in [MaPn81a]. Before discussing these properties, a brief introduction to temporal logic concepts is presented. This is particularly relevant, since the UNITY programming logic is based on a carefully chosen subset of temporal logic [Sand90]. In turn, the UNITY programming logic forms the basis for correctness reasoning in the SLOOP method.

One of the most important aspects of the SLOOP method is the fact that reasoning about correctness is simplified as a result of the computational model being used. As stated in Chapter 1, the latter is based on the UNITY computational model. Since the SLOOP syntax, presented in Chapter 4, is loosely based on the UNITY notation, a brief summary of the **UNITY notation** and **programming logic** is appropriate. In Chapter 5 the correctness properties defined by Manna and Pnueli [MaPn81a] are redefined in terms of the SLOOP logical relations.

In UNITY, program structuring is achieved via **union** and **superposition**. These concepts are summarised in Section 2.5.4. The relevance of union and superposition with respect to SLOOP program structuring will be discussed in Chapter 4 and their impact on correctness reasoning in the SLOOP method will be described in Chapter 7.

Most of the literature referenced in this chapter does not explicitly refer to objects. Aspects related to concurrency are described in terms of **processes**. The definitions and descriptions presented in this chapter therefore refer to processes. A process is defined as an executing program, including the current values of the program counter, stack pointer, registers, the data and stack of the program, as well as any other information needed to run the program [Tane92]. "Conceptually, each process has its own virtual CPU" [Tane92].

An **"active object"** refers to an object that is also a process, i.e. it has its own program to execute [Meye97]. **SLOOP objects that contain parallel[1] operations** may also behave like processes. In the discussions below the meaning of the word process is therefore **extended** to refer to such objects as well.

2.2 The software correctness conundrum

One could ask why software correctness warrants so much research. The main reason is that software is complex and therefore difficult to get right. There are many properties that distinguish software engineering from other engineering disciplines. Parnas et al. [PvSK90] list a few differences:

- **The nature of the errors.**

An error lies dormant in the system until the necessary sequence of events is executed to trigger it. Errors are not introduced due to wear. For example, in the case of hardware, errors due to wear-out can be detected by running diagnostic tests.

- **Tolerance.**

In most engineering disciplines it is good enough to get it almost right. This is based on the fact that the effect of an error is directly proportional to the size of the error. This is not true for software. A punctuation error can cause havoc, while it is possible that a major design oversight can be tolerated. Thus far no useful definition of tolerance exists for software.

- **Inconclusive testing.**

A huge number of tests needs to be performed before one can be reasonably confident about the software. However, it still does not mean that there are no errors. This is a result of the fact that the mathematical functions describing the software may contain an arbitrary number of discontinuities. This property gives software its flexibility, but also its complexity.

2.2.1 Validation, *a posteriori* program verification and the "constructive approach"

The above problems are formidable when dealing with sequential programs; they become even more daunting when **concurrency** is introduced. Due to the multitude of ways in which the processes can interact with one another, there is **a myriad of sequences of events that needs to be tested**. System testing, i.e. executing a system in order to check that it meets user requirements, is an **example** of **validation** [Ince93]. Validation is concerned with checking that the product of a phase matches user requirements [Ince93].

[1] Sequential and parallel SLOOP operations were first described in Chapter 1. More detail will be given in Chapter 4.

One alternative to brute force testing is to apply **mathematical verification techniques**. **Program verification** can be defined as follows: If a program P in a given programming language is correct with respect to a specification ϕ in a given specification language, then the program P is verified with respect to the specification ϕ [Fran92]. A program is correct with respect to its specification if it meets all the requirements imposed by its specification [Lott90].

It is important to note the phrase **"with respect to its specification"** in the above definition. Lott suggests that absolute correctness is a myth due to its ultimate dependence on a high-quality specification [Lott90]. The quality of the specification can be defined as the accuracy with which it reflects the specifier's **intentions**. Lott maintains that it is very difficult to determine the quality of a specification. This suggests that **validation** and **verification** should **complement**, rather than replace each other [Meye90].

Yet another approach towards producing high quality software is the **"constructive approach"**, pioneered by Dijkstra in [Dijk76]. An abstract specification is refined repeatedly, culminating in the derivation of a program. As stated in [Meye97], the aim should be to build a correct system from the outset; not to debug it into correctness. This is the approach followed in the SLOOP method. It has several advantages:

- The **specification** can be **validated after each refinement step**, thereby increasing the probability that the final implementation will indeed match the user requirements.
- The program is constructed together with its correctness arguments. **Errors** can therefore be **detected** during the **early** phases of the development lifecycle.
- It promotes a **disciplined** and **systematic** approach towards software development.
- It is not merely a mechanism to check the correctness of the program after it has been designed and implemented, but rather a **design aid during** all the software development phases.
- It promotes the goal of **seamlessness**; the same processes are used throughout the development lifecycle.

Meyer advocates a **conditional approach** towards ensuring correctness [Meye97]. It is not practical to attempt to prove correctness for a complete system, including hardware, operating system and compiler, in a single step. Instead, the system is divided into layers. At each level, the task is to prove the correctness of that layer only, while **assumptions** are made about the correctness of the supporting layers. This concept is also embraced in the SLOOP method, where the SLOOP statements are at a high level of abstraction, assuming that the mappings to the target architectures are correct. A mapping is a reusable artifact and its correctness needs to be checked only once; thereafter its correctness can be assumed when it is reused. The same principle applies to all SLOOP classes.

2.2.2 Categories of correctness properties

In order to produce unambiguous specifications one has to define correctness properties in more rigorous terms. Partial and total correctness are useful properties of sequential programs that are supposed to terminate. A program that satisfies the **partial correctness** property guarantees that if the precondition that restricts the set of input values is satisfied, then the resulting output values will be correct if the program terminates. Note that it does not guarantee termination. **Total correctness**, on the other hand, guarantees termination as well. Formal definitions of these properties are presented in Sections 2.4.4.1 and 2.4.5.1 respectively.

Partial and total correctness properties also apply to concurrent programs that terminate. However, they do not suffice when dealing with continuous or cyclic programs. In fact, if a continuous program such as an operating system terminates, the program is **incorrect** [Bena90]. Infinite computations also never violate partial correctness properties. For such programs safety

and liveness properties are more appropriate. These notions were introduced by Lamport in [Lamp77].

Safety properties describe properties that need to be satisfied for all execution sequences of the program, i.e. they are properties that assert that something bad will never happen. Examples of safety properties are mutual exclusion and absence of deadlock [Bena90]. Absence of deadlock ensures that a situation cannot arise where each process in the system is awaiting some or other event from one of the other processes in the system and therefore the system is "hung". The conditions for deadlock are given in Section 2.4.4.5. Further discussions around the issue of deadlock are presented in Chapters 4 and 8.

Liveness properties deal with events that must occur eventually, i.e. they are properties that assert that something good will eventually happen. Absence of (individual) starvation, where it is ensured that no process is permanently deprived from obtaining a resource, is an example of a liveness property [Bena90].

A third class of properties can also be distinguished, viz. **precedence properties**. These properties are described in terms of safety and liveness operators [MaPn81a]. An example of a precedence property is fair responsiveness. That means that if process A requests a resource before process B does, then the request will be granted to process A before it will be granted to process B. Formal definitions of all of these (and other) properties are given in Section 2.4.

2.3 Modelling concurrency

In the remaining chapters there are many references to the concepts of interleaving, atomicity, interference and fairness. In the sections below definitions of these concepts are given and it is also described how they apply to the SLOOP programs.

2.3.1 Interleaving

A concurrent system can be viewed as a set of sequential processes executing simultaneously. The speed at which the instructions of the various processes are executed could differ greatly from one processor to the next. As a result the instructions from the various processes could overlap in a multitude of ways. However, these processes only affect each other when there is contention (i.e. they compete for a shared resource) or when they communicate (i.e. pass information between them) [Bena90]. Thus, many different computations involving the instructions that do not cause any interaction between the processes will yield the same result.

For the purposes of correctness reasoning it is therefore reasonable to ignore the fact that these instructions may overlap in time. Instead, the execution of these instructions is modelled in terms of an interleaved sequence of instructions. The instructions that do cause interaction between the processes are also ordered (possibly in an arbitrary way). This is because the hardware ensures that the shared resource is accessed by a single process at a time and in the case of distributed systems the underlying protocol ensures that messages are delivered in some (possibly arbitrary) order [Bena90].

Modelling concurrency by interleaving therefore implies the following abstraction: Each process contains a sequence of atomic instructions. At any moment in time only one process is executing an instruction. That instruction is executed to its completion before the next instruction from that process or one of the other processes is chosen arbitrarily.

If a concurrent program is modelled by an **arbitrary** interleaving of atomic (indivisible) instructions of all the processes constituting the program, then it adequately represents its concurrent behaviour if there are no absolute time requirements. Even though the instructions of

various processes may be executed simultaneously, they are assigned an arbitrary order in the mathematical model.

The interleaving model may seem counter-intuitive for concurrent systems; a non-overlapping execution model is used when the purpose of concurrency is to allow the simultaneous execution of instructions of different processes [MaPn81a]. However, this model is merely a **mathematical model** which is chosen to **simplify analysis**. It takes cognisance of the fact that the effect of instructions where no contention or communication is involved is the same irrespective of the order in which they are executed [MaPn81a].

In a SLOOP program there is no notion of processes. A SLOOP program contains a set of parallel statements that can be executed in any arbitrary order. During the implementation phase, each parallel statement could potentially be mapped to a different process. Provided the atomicity of each parallel statement is preserved, the mapping does not affect correctness reasoning, since the same interleaving model that is used to represent concurrency during the implementation phase (in the executable program) is also used to model concurrency during the design phase (in the SLOOP program).

2.3.2 Atomicity and interference

When reasoning about an arbitrary interleaving of process instructions it is important to define the **atomicity** of these instructions. For example, if the level of atomicity is such that the value of a variable may be read and incremented in one atomic instruction, then a program which contains two concurrent processes that both execute an INCR X instruction, performs correctly under any interleaving of the program instructions. However, if the level of atomicity allows only a single access to a variable per instruction (i.e. three separate LOAD, ADD and STORE instructions have to be executed in order to increment a variable) then different interleavings may yield different results [Bena90].

A program has to perform correctly under **all interleavings within the fairness constraint** (the latter is described below). **Interference**, i.e. the many ways in which the concurrent processes can affect one another as illustrated by the above example, has to be taken into consideration by the designer. For example, if a finer grain of atomicity is defined, then the program becomes more complex (e.g. the designer may have to introduce semaphores to guarantee mutual exclusion of critical sections). The effective interleaving of the instructions is therefore controlled by the program itself by ensuring that certain statements are disabled under certain circumstances. If a process is scheduled, the next instruction is only executed if it is enabled, otherwise the scheduler selects another process.

In the **SLOOP method** the level of **atomicity** is defined as the **parallel statement**. Since the latter has considerable expressive power (as will be evident from Chapter 4), this facilitates software design at a **high level of abstraction**. In turn, it **simplifies correctness reasoning**. This claim will be discussed in more detail in Section 4.3.6.4.

2.3.3 Fairness

As far as the process scheduler is concerned, the only constraint on the interleaving of instructions is the preservation of fairness. Ben-Ari [Bena90] lists the following degrees of fairness:

Weak fairness: A weakly fair system will eventually grant a request made by a process if that process continuously issues the request.

Strong fairness: A strongly fair system will eventually grant a request made by a process if that process issues the request infinitely often.

Linear waiting: Such a system guarantees that a request from a process is serviced before a second request from any other process is serviced, i.e. a request may be overtaken by other processes, but only once.

First In First Out (FIFO): This system guarantees that requests are served on a first come first served basis.

It is useful to have the weaker forms of fairness, especially in distributed systems. For example, it could be very complicated to determine whether a request had been issued **later** than another request in a distributed system. Depending on the application, a weaker form of fairness may suffice.

Manna and Pnueli define both **fairness and justness** [MaPn81a]. The latter deals with the scheduling of individual processes whereas fairness deals with the scheduling of combinations of processes. Justness is sufficient when programs do not contain any semaphores, i.e. all processes are continuously enabled. The scheduling is therefore just if every process is scheduled infinitely often. Fairness has a stronger requirement in order to allow for programs that contain semaphores: it requires that a process that is willing to communicate is eventually scheduled when the states of its communication partners are such that this process can make progress. Thus, each process that is enabled infinitely often, must eventually be scheduled when it is also enabled. An even stronger fairness requirement is to ensure that such progress is made within a finite time.

For example, if two processes A and B communicate via a semaphore in order to ensure the integrity of a critical section, then it will not be fair if process B is scheduled every time when process A has access to the critical section, because process B will then never make any progress. If a program does not contain any semaphores, the notions of justness and fairness are equivalent.

The **SLOOP fairness requirement** states that **each parallel statement has to be executed infinitely often**. Since each statement is always enabled this is a justness or weak fairness requirement.

Modelling a concurrent system as an arbitrary interleaving of the statements of the constituent processes within the fairness constraint therefore provides an abstraction that allows reasoning about the correctness of the system. It is an appropriate abstraction for all concurrent systems except those with absolute time requirements, because no assumptions can be made about the sequence in which statements from the various processes will be executed. In the case of real-time systems, modifications or extensions to the interleaving model may be required in order to indicate the time dependencies [Bena90]. By defining the parallel statement as the atomic unit of execution in a SLOOP program, the designer is given the capability to prevent undesirable interference at a fairly high level of abstraction.

The notion of a program comprising of several processes that are all scheduled infinitely often, is even suitable to model a program that terminates. The latter is just a special case of a program which executes ad infinitum. Termination is described as the state which a program reaches where the execution of any statement does not change the state of the program.

2.4 Formulating correctness properties

Whether reasoning about the correctness of a program occurs *a priori* as in the "constructive approach" or *a posteriori* as in program verification, the correctness properties need to be formulated in a precise, unambiguous way.

Many formalisms exist which serve this purpose, for example graphical notations and logic-based formulas. Of particular interest is the use of **temporal logic** in this regard, since the UNITY proof theory was heavily influenced by this type of logic [ChMi88].

The next section serves as a brief introduction to temporal logic in order to provide a sufficient backgound to the concepts involved.

2.4.1 Temporal logic

The introduction of temporal logic as a verification tool for concurrent programs is attributed to Pnueli [Pnue77]. Temporal logic is a specific type of **modal logic**. In [MaPn81a], Manna and Pnueli provide an exposition of the evolvement of modal logic from predicate calculus, which, in turn, stems from propositional calculus.

A **propositional** formula describes a static, constant property which may be either true or false. The formula may have one or more constituent parts. Manna and Pnueli cite the example, "it rains today" as a propositional formula which is either true or false [MaPn81a]. **Predicate** calculus allows the truth or falsity of a formula to vary depending on the values of certain parameters. Thus, a formula is written in terms of a predicate and its parameters. The above example can therefore be rewritten as *rain(l,t)*, where l represents the location and t represents the day. Depending on the day and the location (i.e. the values of the parameters) the formula may be either true or false.

Modal logic adds another dimension to the formula. Not only does its truth or falsity depend on the parameters of the predicate, but the meaning of the predicate itself may be changed, depending on the **mode** of the formula. For example, the formula *rain(x)* may be interpreted in terms of the universe of time, i.e. time is the implicit factor. Given that time is a certain day, *rain(x)* states whether it rained on that day at a certain location x. Similarly, location could be made the implicit factor and time could be made the explicit parameter. Thus, the mode could be location instead of time. Temporal logic is a modal logic where **time** is the **implicit factor**.

Although it is possible to write temporal formulas in terms of predicate calculus, temporal logic provides a natural, convenient and concise notation to describe dynamic behaviour of programs.

The advantage of using temporal logic is evident from the following example [Mosz86]:

In a system which has time-dependent variables, classical logic can be used to describe the system by modelling these variables as explicit functions of time. The following formula describes an interval of time in which the variable I at some time t equals 1 and at some later time t' equals 2.

$$\exists\, t,t' :\ (\,[\,t < t'\,] \wedge [I(t)=1] \wedge [I(t')=2]\,)$$

Although this is a powerful method of specifying time-dependent variables, it suffers from a proliferation of time variables and quantifiers.

The following formula contains the "eventually" temporal operator $\Diamond$:

$\Diamond\ [\ (I{=}1) \Rightarrow \Diamond\ (I{=}2)\ ]$

The semantics are the same as for the first formula. The construct $\Diamond w$ is true if there is some suffix subinterval in which the formula w is true.

Manna and Pnueli [MaPn81a] contend that predicate logic adequately describes static situations, but **dynamic behaviour** is more conveniently described by modal logic, and more specifically temporal logic. The execution of a program can be viewed as the continuous changing from one state to another [Krög87]. The execution of these states plays the role of time. In each state one or more formulas may be true or false. Thus, the truth or falsity of the formulas depends on the state.

2.4.2 Temporal operators

As explained in the previous section, temporal logic is an extension of classical logic using a number of temporal operators. Some of the most important temporal operators are $\Box$, $\Diamond$, O and U. The meaning of these operators are as follows (where ω, ω_1 and ω_2 are well-formed formulas in temporal logic):

$\Box\omega$	ω holds at all time points after the reference time point (the **always** operator)
$\Diamond\ \omega$	ω holds at some time point after the reference time point (the sometime or **eventually** operator)
O ω	ω holds at the next time point after the reference time point (the **next time** operator)
$\omega_1\ U\ \omega_2$	ω_1 holds until the instant that ω_2 becomes true but not including that instant (the **until** operator)

The reference time point may be chosen arbitrarily.

A well-formed formula comprises atomic formulas (propositions or predicates) to which boolean connectives, the existential and universal quantifiers ($\exists$ and $\forall$ respectively) and temporal operators have been applied [MaPn81a].

The temporal logic operators that have been discussed thus far refer to the present and the future. There have also been some linguistic extensions to temporal logic to include operators that deal with the past [Wolp87], but these are not relevant to the present discussion. Numerous temporal logic variants have been developed [Wolp87, Mosz86], many of them targeting specific types of problems. For example, some contain constructs that facilitate an elegant representation of non-determinism, others are particularly appropriate for distributed systems. These variants are not described in more detail here, since they are not relevant to the SLOOP method.

2.4.3 Which correctness properties?

Generally the approaches towards system specification are broadly classified as logic-based and model-based [JiZh96]. Temporal logic and finite state machines are examples of the respective formalisms. The conjunctive nature of a logic-based method raises the question of the **completeness** and **consistency** of the specification, i.e.

- Have all the relevant properties been specified (completeness)?
- Are there any contradictory properties (consistency)?

It is obvious that a specification has to be consistent. However, completeness is a more difficult issue. Meyer [Meye97] argues that in order to measure the completeness of a specification, it is necessary to compare it against a reference document. If the requirements in the latter are specified informally, the completeness of the specification cannot be checked systematically. If the contents of the reference document is formal, it merely elevates the problem to the next level:

the completeness of the reference document becomes the issue. Francez [Fran92] mentions that it is not possible to guarantee that a specification indeed reflects the specifier's intentions, since intentions are mental objects.

There is also a delicate balance between overspecification and underspecification. The specifier should ensure that enough is said to prevent an unacceptable implementation from being chosen, without resulting in implementation bias [Wing90].

The following list of correctness properties serve as a **sample** of useful properties that can be considered in a specification [MaPn81a]:

Invariance (safety) properties:
a) Partial correctness
b) Clean behaviour
c) Global and local invariants
d) Mutual exclusion
e) Deadlock freedom
f) Generalized deadlock freedom

Eventuality (liveness) properties:
a) Total correctness
b) Intermittent assertions
c) Accessibility
d) Liveness
e) Responsiveness

Precedence (until) properties:
a) Safe liveness
b) Absence of unsolicited response
c) Fair responsiveness

Note that the above list is not exhaustive, but is used as a convenient checklist in SLOOP specifications. The definitions of the properties that are given below are from [MaPn81a]. In Chapter 5 they are rewritten in terms of the logical relations used in the SLOOP method and in Chapter 7 they are exemplified in the SLOOP context.

The following symbols are used in the definitions of the correctness properties of some program P:

$\varphi(\bar{x})$	The precondition that restricts the set of inputs $\bar{x}$ for which P is correct.
$\psi(\bar{x},\bar{y})$	The statement of correctness, i.e. the relation that should hold between the input values $\bar{x}$ and the output values $\bar{y}$.
$\models \omega$	The formula ω with respect to P and φ is valid. (A formula is valid with respect to P and φ if it is true for the set of all computations of P whose input complies with φ.)

The formulas expressing the safety properties below have the form $\models \omega$. They are written as $\models \Box \omega$ to emphasise the invariant character of these properties. If it is necessary to emphasise the precondition $\varphi(\bar{x})$, the formulas are written as

$\models \varphi(\bar{x}) \supset \Box\omega$.

In all the temporal logic formulas the $\supset$ symbol indicates implication.

2.4.4 Safety properties

The subsections below give definitions of the safety properties listed earlier.

2.4.4.1 Partial correctness

Partial correctness is only meaningful for terminating programs (non-terminating programs are always partially correct). It can be represented by the following statement:

$\models \quad \varphi(\bar{x}) \supset \Box(\text{at } \bar{l}_e \supset \psi(\bar{x}, \bar{y}))$

where $\bar{l}_e$ is the vector of the terminal locations in all the processes.

It states that if the preconditions restricting the input of the program are satisfied, then the correctness statement is always satisfied if the program reaches a terminating state. Partial correctness does not guarantee that the program will terminate. It only states that the program will be correct if it does terminate.

2.4.4.2 Clean behaviour

If a program exhibits clean behaviour, it means that the execution of any statement will not result in an exception condition. For each location l in the program a cleanness condition α_l can be defined. For example, the cleanness condition of a statement which involves division, requires the divisor to be non-zero. If a statement references an element of an array, the cleanness condition specifies that the array subscript should be within the declared range.

The clean behaviour of the program is specified by the following statement:

$\models \quad \varphi(\bar{x}) \supset \Box \bigwedge_l (\text{at } l \supset \alpha_l).$

The conjunction is taken over all locations where exceptions could potentially occur.

2.4.4.3 Global and local invariants

A global invariant refers to a program property that holds throughout the computation, i.e. it is independent of the program location. It is written as:

$\models \quad \varphi(\bar{x}) \supset \Box\beta.$

A local invariant refers to a program property that holds at a particular location. If this location is an exit location, the property becomes a partial correctness property, i.e. partial correctness is a special case of local invariance. Local invariance is written as

$\models \quad \Box(\text{at } l \supset \beta).$

2.4.4.4 Mutual exclusion

When two or more processes execute concurrently, it may be necessary to ensure that certain sections of their code will never be executed simultaneously, e.g. when they access a shared resource. Such a section of code is called a critical section. The property stating that such critical sections will never be executed simultaneously, is called mutual exclusion. It is represented by:

$\models \quad \varphi(\bar{x}) \supset \Box\neg(\text{at } C_1 \wedge \text{at } C_2)$

where C_1 and C_2 are the critical sections of processes 1 and 2 respectively.

2.4.4.5 Deadlock freedom

Concurrent processes may communicate with each other by exchanging messages or by sharing resources. A waiting location is a location where a process is waiting for an action or message from another process. These are the only **potential** deadlock locations [MaPn81a]. The formal definition of a waiting location l specifies that the full-exit condition E_l is not identically true.

A full-exit condition at location l is the logical *or* of all the enabling conditions at location l. Such an enabling condition, $c_i(\bar{y})$, is a boolean function c_i of the values of the variables $\bar{y}$ shared between the processes. If $c_i(\bar{y})$ is true, then control may continue beyond location l. The full-exit condition of location l can therefore be written as $E_l(\bar{y}) = c_1(\bar{y}) \vee \ldots \vee c_k(\bar{y})$, where $\bar{y}$ represents all the shared variables. A full-exit condition is identically true if it is true for every $\bar{y}$. If more than one enabling condition is true (i.e. several transitions are enabled), then a non-deterministic choice between possible transitions may be made.

For example, the full-exit condition of the "loop until $\neg p(\bar{y})$" statement is identically true, because whenever the statement is scheduled, either the loop instruction or the escape instruction is enabled ($E_l(\bar{y}) = p(\bar{y}) \vee \neg p(\bar{y})$). The full-exit condition for the "wait until $p(\bar{y})$" statement, on the other hand, is not identically true, since the full-exit condition has to evaluate to $p(\bar{y})$ for the statement to be enabled. It may happen that the full-exit condition evaluates to $\neg p(\bar{y})$ whenever the statement is scheduled. It is for this reason that deadlock is possible if the full-exit condition is not identically true.

Should all the communicating processes be at a waiting location and no process is enabled, then deadlock has occurred, i.e. no progress can be made. If there are m processes and the tuple $l = (l^1, \ldots, l^m)$ represents the waiting locations in these processes, then the following statement describes deadlock freedom with respect to l:

$$\models \quad \varphi(\bar{x}) \supset \Box\Big(\bigwedge_{j=1}^{m} \text{at } l^j \supset \bigvee_{j=1}^{m} E_j(\bar{y})\Big).$$

It means that if all processes are at waiting locations, then the full exit condition of at least one process will be true, i.e. at least one process will be enabled. A program is only deadlock free if the above property is true for all possible combinations of waiting locations except where all the processes are at their terminating locations.

2.4.4.6 Generalised deadlock freedom

Generalised deadlock freedom has an even stronger requirement than deadlock freedom. The full-exit conditions at the waiting locations are generalised to include looping instructions. This implies that although a full-exit condition of a process may be true, the process may not be able to make any progress, since it is only executing a looping instruction. Generalised deadlock freedom therefore requires that at least one of the processes has to have an **escape** condition enabled when the processes are at their waiting locations. Thus, if the escape condition $\varepsilon_j(\bar{y})$ is true, it represents an exit condition that ensures progress from location l^j.

Generalised deadlock freedom is therefore represented as:

$$\models \quad \varphi(\bar{x}) \supset \Box\Big(\bigwedge_{j=1}^{m} \text{at } l^j \supset \bigvee_{j=1}^{m} \varepsilon_j(\bar{y})\Big).$$

Again, a program is only free from generalised deadlock if the above formula is true for all combinations of waiting locations except where all the programs are at their terminating locations.

2.4.5 Liveness properties

The subsections below present definitions of the liveness properties listed earlier.

2.4.5.1 Total correctness

Total correctness specifies that a program will terminate, and when it terminates, the correctness statement will be satisfied. This is expressed as follows:

$|\equiv \quad \varphi(\bar{x}) \supset \Diamond\ (\text{at } \bar{l}_e \wedge \psi(\bar{x}, \bar{y}))$.

In other words, eventually an exit location will be reached, and when that happens, the results will be correct.

2.4.5.2 Intermittent assertions

An intermittent assertion describes the relation between two events that may occur during program execution. For example, if ϕ holds at location l, then eventually location l' will be reached, where ϕ' will hold. It is written as:

$|\equiv \quad (\text{at } l \wedge \phi) \supset \Diamond (\text{at } l' \wedge \phi')$.

The intermittent assertion property is especially important when dealing with cyclic (continuous) programs, since there is no exit location with the corresponding termination correctness statement.

2.4.5.3 Accessibility

The accessibility property specifies that a process that needs to enter its critical section will eventually be able to do so. If l_1 represents the location just before entering the critical section and C represents the critical section, then accessibility is expressed by the statement:

$|\equiv \quad \text{at } l_1 \supset \Diamond \text{at } C$.

The correct construction of a critical section should ensure that both the accessibility and mutual exclusion properties are satisfied, since these are complementary properties.

2.4.5.4 Liveness

A process exhibits the liveness property if it can be stated that if the process is at location l, where l is not an exit location, it will eventually move to another location, i.e.

$|\equiv \quad \neg \Box \text{ at } l$.

Liveness is also known as freedom from individual starvation. This is a stronger requirement than generalised deadlock freedom, because it specifies that an individual process must progress. In the case of generalised deadlock freedom it is only specified that at least one process must progress.

2.4.5.5 Responsiveness

In the case of cyclic programs, partial and total correctness properties are meaningless, since the programs do not terminate. Cyclic programs usually have to respond to events such as requests for resources or action. It must be ensured that a request will eventually be granted, i.e.

$|\equiv \quad r_i \supset \Diamond g_i$

where r_i represents request i and g_i represents granting request i.

2.4.6 Precedence properties

The subsections below deal with the precedence properties listed earlier.

2.4.6.1 Safe liveness

Safe liveness properties are expressed as follows:

$|\equiv \quad \omega_1 U \omega_2.$

Thus, ω_1 holds up to but not including the instant when ω_2 starts to hold.

Manna and Pnueli [MaPn81a] state that the full specification of a program can be expressed in terms of an *until* expression, in other words, the end result ω_2 will eventually be achieved (liveness) while maintaining the safety properties ω_1.

2.4.6.2 Absence of unsolicited response

The absence of unsolicited response specifies that ω_2 will never happen unless preceded by ω_1. It is represented by:

$|\equiv \quad \omega_1 P \omega_2$

where the precede operator P is derived from the until operator U in the following way:

$\omega_1 P \omega_2$ is $\neg((\neg \omega_1) U \omega_2)$.

However, if a situation can occur where ω_1 is true at t_1 and ω_2 is true at t_3, but neither is true at t_2, where $t_1 < t_2 < t_3$, then the formula will be false if the reference point is t_2. In that case the following formula ensures that the reference point is chosen correctly:

$|\equiv \quad (\text{at } l_0 \supset \omega_1 P \omega_2) \wedge [(\omega_2 \wedge \mathrm{O} \neg \omega_2) \supset \mathrm{O}(\omega_1 P \omega_2)].$

Thus, the reference point is either the starting point of the computation or an instant in which ω_2 is true and becomes false in the following instant. In the latter case $\omega_1 P \omega_2$ begins to hold in the next instant.

If it is known that once ω_1 becomes true, it continues to hold until ω_2 becomes true, the following formula suffices:

$|\equiv \quad (\text{at } l_0 \vee \neg \omega_1) \supset (\omega_1 P \omega_2).$

2.4.6.3 Fair responsiveness

Fair responsiveness states that ψ_1 will only precede ψ_2 if two earlier events ϕ_1 and ϕ_2 occurred in the same order. This is used to specify that if a request from process A arrives before a request from process B, then the request from process A will be granted before the request from process B is granted. This requires a conditional precedence statement together with responsiveness statements. The conditional precedence statement is written as

$\models \equiv \quad (\phi_1 P \phi_2) \supset (\psi_1 P \psi_2)$

and the responsiveness statements are given as
$\models \equiv \quad (\phi_1 \supset \Diamond \psi_1)$ and $\models \equiv \quad (\phi_2 \supset \Diamond \psi_2)$.

2.5 UNITY

As stated in Chapter 1, the SLOOP method is loosely based on UNITY, therefore a brief description of the latter is appropriate. UNITY is a **computational model** and **proof system** described in the book titled "Parallel program design" by Chandy and Misra [ChMi88]. They emphasise that the method which they propose is not restricted to concurrent programs. First and foremost they focus on the problem solving task. The target architecture and implementation language are secondary concerns. As a result their method applies to the whole spectrum of sequential and concurrent programs. This concept of first developing a centralised solution, which is then transformed into a distributed solution via a sequence of correctness preserving steps is also promoted by Back and Kurki-Suonio [BaKu89].

The UNITY theory is based on what Chandy and Misra consider to be the fundamental aspects of programming. Amongst these are issues such as **non-determinism**, the **absence of control flow**, **assignments** and the **state-transition model**, hence the name UNITY, which is an acronym for Unbounded Nondeterministic Iterative Transformations. The name also reflects their view that programming problems should be solved in a **unified** manner before considering the target architecture.

2.5.1 Non-determinism

At the highest level of abstraction a **non-deterministic** solution is provided. For systems that are inherently non-deterministic, further refinements of the solution retain the non-determinism. For other types of systems, the non-determinism may be limited during further refinements by disallowing certain executions at each step.

2.5.2 Assignments and absence of control flow

One of the main characteristics of a UNITY program is the **absence of control flow,** i.e. there is no notion of a program location counter. Each program comprises a section where the variables are declared, an initialisation section and a number of multiple assignment statements. A multiple assignment statement may be conditional and is executed infinitely often. The order in which the statements are executed is irrelevant, as long as the **fairness constraint** is satisfied, i.e. the statements are all executed infinitely often. The assignment components that comprise a single multiple assignment statement are executed simultaneously. Each multiple assignment statement is executed **atomically**.

The structure of a UNITY program is given below using BNF [ChMi88]. All non-terminal symbols are presented in italics. Plain or boldface type designates a terminal symbol. Syntactic units that may occur zero or more times are enclosed in braces.

program	→	**Program**	*program-name*
		declare	*declare-section*
		always	*always-section*
		initially	*initially-section*
		assign	*assign-section*
		end	

The *declare-section* contains the names and associated types of the variables used in the program. The *always-section* is optional. It is used to define certain variables as functions of others and is a notational convenience.

The *initially-section* has the same syntax as the *assign-section*, except that the assignment symbol is replaced by the equals symbol. This is because the *initially-section* should be treated as the specification of the predicate that holds initially. The order of the equations in this section is important. Variables that appear on the left-hand side of an equation may only be referenced on the right-hand side of **subsequent** equations.

The syntax of the *assign-section* is given below using BNF [ChMi88]:

assign-section	→	*statement-list*
statement-list	→	*statement* {[] *statement* }
statement	→	*assignment-statement* | *quantified-statement-list*
quantified-statement-list	→	<[] *quantification statement-list*>
quantification	→	*variable-list* : *boolean-expr* ::
assignment-statement	→	*assignment-component* { || *assignment-component* }
assignment-component	→	*enumerated-assignment* | *quantified-assignment*
enumerated-assignment	→	*variable-list* := *expr-list*
quantified-assignment	→	<|| *quantification assignment-statement*>
variable-list	→	*variable* {, *variable* }
expr-list	→	*simple-expr-list* | *conditional-expr-list*
simple-expr-list	→	*expr* {, *expr* }
conditional-expr-list	→	*simple-expr-list* if *boolean-expr* {~ *simple-expr-list* if *boolean-expr* }

PASCAL-style expressions and boolean expressions are implied in the case of *expr* and *boolean-expr* respectively. Comments are enclosed in braces.

Assignment statements are separated by the [] symbol. In turn, each assignment statement may consist of multiple **assignment components** separated by the || symbol. Assignment components may be **conditional**. Multiple boolean expressions may be associated with a conditional assignment component. Each boolean expression with its associated expression list is separated from the others within the same assignment component via the ~ symbol. If none of the boolean expressions evaluates to true, the values of the variables remain the same; if multiple boolean expressions are true, then the assignment associated with each true expression should result in the same values [ChMi88].

Thus, although the selection of the next multiple assigment statement to execute is performed non-deterministically within the fairness constraint, each assignment statement is deterministic; only one result is possible in a given state.

It is important to note that all subscripts, as well as the right-hand side of **each** assignment component of a multiple assignment statement, are evaluated **before** the variables on the left-hand side of the statement receive their new values.

The following example [ChMi88] illustrates the multiple assignment statement syntax:

```
   z := 2 * z      if y ≥ 2 * z ~
        N          if y < 2 * z
|| k := 2 * k      if y ≥ 2 * z ~
        1          if y < 2 * z

[]  x := x + k     if y ≥ z
||  y := y - z     if y ≥ z
```

This program divides M by N and it stores the quotient in x and the remainder in y if the variables are initialised as follows: $x = 0$, $y = M$, $z = N$ and $k = 1$.

There are two multiple **assignment statements** in the above example, separated by the [] symbol. Only one such statement is executed at a time. These statements can be executed in any order. The only fairness requirement is that each multiple assignment statement should be executed infinitely often. Each assignment statement in the above example has two **assignment components** separated by the || symbol.

An operational view of the assignment statement execution is that the **components** of such a statement execute **simultaneously**. Thus, the expression on the right-hand side of each assignment component (and all subscripts on the left-hand side of each assignment component) are evaluated first. Only once all these evaluations have been completed, the resulting values are assigned to the variables on the left-hand side of the respective assignment components. For example, in the first statement above, all the conditions are evaluated first. If the condition $y \geq 2 * z$ evaluates to true, then both the expressions $2 * z$ and $2 * k$ are evaluated before the variables z and k are updated with the respective values. If the condition $y < 2 * z$ evaluates to true, then the variables z and k are updated with the values N and 1 respectively. The tilde symbol is used to separate the alternatives within a statement component.

The above statements can also be written more succinctly as:

```
   z, k := 2 * z, 2 * k      if y ≥ 2 * z ~
           N, 1              if y < 2 * z
[]  x, y := x + k,  y - z     if y ≥ z
```

The syntax of the *assign-section* allows for both **quantified** assignment statement lists and quantified assignments, i.e. generic statements that are instantiated over all possible values of the quantified variables appearing in the statements. The scope of a quantification is delineated by the brackets "<" and ">".

The following example [ChMi88] shows the usage of both a *quantified-statement-list* and a *quantified-assignment*:

```
< [] i: 0 ≤ i ≤ N ::
        U[i,i] := 1
      [] < || j: 0 ≤ j ≤ N ∧ i ≠ j :: U[i,j] := 0 >
>
```

This example has `N + 1` statement lists, each containing an enumerated assignment and a quantified assignment. The former assigns the value one to the diagonal element of a matrix designated by i and the latter assigns the value zero to the off-diagonal elements in row i of the matrix.

2.5.3 State transitions

When an assignment is executed, the values of the variables may change, i.e. the program **state** is modified. These are the iterative transformations that the acronym UNITY refers to. If the program state is unchanged, no matter which statement is executed, then the program has reached a **fixed point**. This is equivalent to program **termination** in the case of a stand-alone sequential program. The fixed point of a program that runs concurrently with others is defined as a state which can be changed only by other programs.

2.5.4 Program structuring

One way of constructing a complex program is to combine several smaller and simpler program components. Chandy and Misra discuss two ways of achieving this, viz. **union** and **superposition**. In the case of program union, the statements of the one program component are merely appended to the other program component. The assignments in the program components may reference common variables. Program union is denoted by F [] G. Superposition (denoted by F') is achieved by applying the **augmentation** and **restricted union** rules. The former specifies that a statement s in the underlying program may be transformed into a statement $s \parallel r$, where r does not assign to any variables in the underlying program. The restricted union rule allows statements to be added to the underlying program, provided they do not assign to any of the variables in the underlying program. The keywords **transform** and **add** denote the statements affected by the two rules respectively.

The following example from [ChMi88] illustrates the superposition technique:

```
Program Detection_example
  initially count, claim = 0, false
  transform
     each statement s in the underlying program to
     s || count := count + 1
  add claim := (count > 10)
end { Detection_example }
```

The underlying program in the above example is transformed as shown above in order to detect the claim that the number of statements executed by a program exceeds 10. Program structuring in the SLOOP method is discussed in Chapter 4. The impact of union and superposition on correctness reasoning is considered in Chapter 7.

2.5.5 A logic for the specification, design and verification of UNITY programs

The lack of control flow in UNITY programs contributes significantly towards simplifying verification of UNITY programs.

The conventional way of reasoning about the correctness of a sequential program is to associate predicates with specific locations in the program text. This becomes exceedingly difficult when concurrency is introduced, due to the many possible interleavings of the statements from the various processes constituting the concurrent program. (Examples of correctness proofs based on temporal logic that include location counters can be found in [MaPn81b].) It is evident from the above that there is no notion of a program location counter in a UNITY program. Any statement can be scheduled for execution at any time, provided the fairness constraint is satisfied. The correctness properties are therefore not associated with the position of the program location counter, but with the program as a whole.

The notation $\{p\}\ s\ \{q\}$, where p and q denote the precondition and postcondition for statement s respectively, was first introduced to associate predicates with the statements of sequential programs. The assertions associated with UNITY programs have the same format, but instead of

referring to statements at specific locations of control flow, the statement s is **universally or existentially quantified** over the statements of the entire program [ChMi88]. It is important that the statement s should terminate, otherwise the fairness requirement cannot be satisfied.

The semantics of UNITY assignment statements are given in [ChMi88]. The basic principle is as follows: if a statement s contains an assignment of expression E to variable x, then in order to prove $\{p\}\ s\ \{q\}$, one has to prove that p implies q if E is substituted for all occurrences of x in q. In the case of multiple assignment statements the expressions on the right-hand side of the assignment components are substituted **simultaneously** for the corresponding variables on the left-hand side of the assignment components. In a quantified assignment statement, simultaneous assignments may be made to an array of variables. In [ChMi88], Chandy and Misra show how the precondition of a quantified statement is obtained from its postcondition by extending the notation of assignment to subscripted variables to allow quantification over array subscripts.

The UNITY proof system[2] is based on three fundamental **logical relations**, viz. *unless*, *ensures* and *leads-to*. Their definitions [ChMi88] are given below. A statement in a given program F is denoted by s.

For a given program F
p *unless* $q \equiv \langle \forall s: s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{p \vee q\}\rangle$
Thus, either q never holds and p continues to hold forever, or q holds eventually and p holds at least until q holds.

For a given program F
p *ensures* $q \equiv (p \text{ unless } q \wedge \langle \exists s: s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{q\}\rangle)$
Thus, if p holds at some point in F, then p remains true as long as q is false and eventually q becomes true. It is important to note that this relation refers to the existence of a **single** statement that establishes q when $p \wedge \neg q$ holds.

The *leads-to* relation (indicated by $p \rightarrow q$) specifies that once p is true, q is or will be true. A program has the property $p \rightarrow q$ if and only if it can be derived by a finite number of applications of the inference rules below. The meaning of an inference rule is that the formula below the line (the conclusion) is true only if the formulae above the line (the premises) are true.

- $\dfrac{p \text{ ensures } q}{p \rightarrow q}$

- $\dfrac{p \rightarrow q,\ q \rightarrow r}{p \rightarrow r}$ (transitivity)

- For any set S, $\dfrac{\langle \forall p: p \in S :: p \rightarrow q \rangle}{\langle \exists p: p \in S :: p \rangle \rightarrow q}$ (disjunction)
  where S is any set of predicates.

The disjunction rule is from [Misr99]. It states that if for all $p \in S$ it is true that $p \rightarrow q$, then if any one of the predicates $p \in S$ is true, q will eventually hold.

[2] There have been some changes [Misr99] to the UNITY logic since its publication in [ChMi88], but for the purposes of its application in the SLOOP method, the original definitions suffice.

The *leads-to* relation allows for transitivity, i.e. if $p \rightarrow r$ and $r \rightarrow q$, then $p \rightarrow q$. The main difference between the *ensures* and *leads-to* relations is the fact that it cannot be asserted that p holds as long as q does not.

The following properties are special cases of *unless* (the $\Rightarrow$ symbol denotes implication):

- *stable* $p \equiv p$ *unless false*
- *invariant* $p \equiv$ (initial condition $\Rightarrow p$) $\wedge$ *stable* p

The *unless*, *stable* and *invariant* relations are used to specify safety properties, whereas the *ensures* and *leads-to* relations denote progress properties.

The above relations represent the fundamental concepts of the UNITY programming logic. Other useful relations may be derived from them, for example, the *detects* and *until* relations. They are defined as follows:

p *detects* $q \equiv (p \Rightarrow q) \wedge (q \rightarrow p)$
p *until* $q \equiv (p$ *unless* $q) \wedge (p \rightarrow q)$
p *precedes* $q \equiv \neg((\neg p)$ *until* $q)$

If p *detects* q, then it means that p holds within a finite time of q holding and once p holds, then so does q. The difference between *ensures* and *until* is that **exactly one** statement establishes q in the case of *ensures*.

The fixed point predicate *FP* is defined as follows:
$FP \equiv < \forall s: s$ in $F \wedge s$ is $X := E :: X = E >$

Thus, for all statements in program F, the value of the expression on the right-hand side of a statement is equal to the value represented by the variable on the left-hand side. When a stand-alone sequential program reaches a fixed point, it is equivalent to termination.

The **UNITY programming logic** is based on a carefully chosen **fragment of linear temporal logic**. The concept of O (*next state*) is not used, but it is clear from the above that the *unless* relation corresponds with the *weak until* temporal operator and the *ensures* and *leads-to* relations represent the *eventuality* operator.

The distinctive feature of a UNITY assertion is that it contains **no references to location counters**. In Chapter 7 it is shown how the SLOOP method takes advantage of such an approach.

2.6 Summary

This chapter served as an **introduction** to the **software correctness aspects** of the SLOOP method. It summarised the basic concepts found in the literature that are related to this topic, and which are also specifically applicable to the SLOOP method.

Software correctness can be defined as the compliance of software with specified correctness properties. This is a vast topic. This chapter covered those aspects that are particularly relevant to the SLOOP method, viz.

- why the specification of correctness properties is important,
- the way in which they are applied,
- the terminology used to describe them,
- which correctness properties to consider,
- their applicability to concurrent systems and
- their ease of use.

First of all it was argued that it does not suffice to rely **solely** on the testing of an implementation (i.e. a form of validation) to ensure that a reliable and functionally correct product is produced. Not only is it not possible to guarantee an absence of errors, but such an approach usually also suffers from a lack of seamlessness. Often the design and testing phases have nothing more in common other than the product of the analysis phase, viz. an informal (i.e. natural language) specification of the required functionality of the system, which serves as basis for the design and test case documents.

In contrast, the SLOOP method embraces the concept of designing a system correctly from the outset rather than debugging it into correctness. In order to achieve this, the emphasis is on the **specification of a set of correctness properties**. These properties are **refined** repeatedly until a SLOOP program can be **derived**. Finally, a simple **mapping** is required in order to produce an executable program.

Thus, the focus is on the correctness properties throughout the software development lifecycle. The correctness properties may be specified informally during the requirements analysis phase. During the design phase they are refined into a more formal notation and the SLOOP program statements are derived. During the implementation phase these statements are mapped to the target architecture, thereby achieving a seamless transformation of the specification of correctness properties into an executable program.

The specification of correctness properties is therefore important in order to be able to follow the **"constructive approach"** towards software development and it is essential in order to support reasoning about the correctness of the system.

The next issue to consider is the **representation of the correctness properties**. Many formalisms exist, but in this chapter the focus was on temporal logic, since it forms the basis of the logic used in UNITY and therefore also in the SLOOP method. A brief introduction to temporal logic was followed by a set of definitions of correctness properties in terms of temporal logic.

Although not exhaustive, this set of correctness properties serves as a **useful checklist** during system development. It is noted that it is difficult to ascertain the completeness of a logic-based specification, due to the conjunctive nature of such a specification. Furthermore, it needs to be measured against the intentions of the specifier. Since these intentions are mental objects, it cannot be checked formally. This is one of the reasons why validation still has a part to play in the software development process.

One of the advantages of the "constructive approach" is the fact that validation can be performed on the product of each refinement step, not only on the final implementation. During the analysis phase, the requirements are specified in terms of a set of properties, albeit informally. That specification can be presented to the user to validate that it describes the required behaviour of the system. During the design phase the correctness properties are refined and specified in a formal notation. This facilitates the validation of the behaviour of the system based on an unambiguous and precise specification. The support of rapid prototyping during the design phase also facilitates validation of the design. This enhances the possibility of detecting errors during the early phases of the software development lifecycle.

The last part of this chapter was devoted to a brief introduction to UNITY. It is particularly important because the computational model used in UNITY provides the key to the **simplification of correctness arguments**, since program location counters can be ignored. This is particularly significant when reasoning about **concurrent systems**. Since the same computational model is used in the SLOOP method, this also applies to correctness arguments in the latter.

In Chapter 1 it was stated that the aim of the research was to arrive at a software development method which satisfied a certain set of goals. Several of these goals were addressed in this chapter, viz.

- support for reasoning about correctness,
- seamlessness,
- the promotion of a unified software development approach and
- the need to be usable by practising software designers.

Although the SLOOP method is not formal, it is based on concepts that are associated with formal methods. It provides sufficient rigour in order to **support reasoning about correctness**. This chapter provided the background to concepts such as the "constructive approach", temporal logic, definitions of useful correctness properties and the UNITY computational model. These concepts underpin the SLOOP method, as will be evident from Chapter 4 onwards.

Earlier in this section it was described how the goal of **seamlessness** is achieved by making each software development phase merely an extension of the previous phase in terms of the notation and processes that are used.

The UNITY method focuses on first designing a solution to the problem to be solved; issues such as the target architecture and implementation language are later concerns. The allocation of statements and variables to processes is only performed once a unified solution has been constructed. The SLOOP method also adopts such a **unified software development approach**.

This chapter described how concurrency could be modelled by the interleaving of the instructions of the processes constituting a program. It was shown how the level of atomicity of these instructions could determine the interference that the designer would have to take into consideration.

The core of a SLOOP program is a set of parallel statements executing infinitely often. Each parallel statement executes atomically. These parallel statements have considerable expressive power, as will be shown in Chapter 4. The designer is therefore provided with the capability to design a program at a fairly high level of abstraction, thereby simplifying correctness reasoning. The computational model, which allows one to disregard location counters, together with the structuring capabilities provided by the object-oriented approach, further simplifies correctness reasoning. Such simplifications contribute towards the goal of making the method **usable by practising software designers**.

The correctness reasoning is informal, therefore there is no need to be conversant with the theorems of the programming logic. Experience with the SLOOP method has shown that by merely assigning such an important role to correctness reasoning, albeit informal, already contributes towards producing software systems that are more correct from the outset. By following the "lightweight" approach, the method is usable by practising software designers, while one still reaps the benefits of the emphasis on correctness reasoning.

All of these issues will become more evident from Chapter 5 onwards, where a non-trivial case study is used to illustrate various aspects of the SLOOP method.

CHAPTER 3

THE OBJECT-ORIENTED PARADIGM

3.1 Introduction

The SLOOP method is an object-oriented software development method. It therefore benefits from the inherent **structuring** capabilities of object-orientation. It also takes advantage of its **reusability** features.

This chapter contains the second part of the literature study. It provides an introduction to those aspects of the object-oriented paradigm that are relevant to the SLOOP method. The following topics are covered:

- object-orientation and concurrency,
- design patterns and frameworks,
- the specification of object interaction and
- object-oriented formal methods.

The concept of object-orientation is well suited to concurrency. As shown in the previous chapter, the initial work in the area of concurrency was based on the notion of a process. Meyer lists some similarities between processes and objects [Meye97]:

- Both are autonomous, encapsulated software artifacts.
- Interprocess communication is facilitated via a few well-defined mechanisms. Similarly, the operations of an object present a well-defined interface for interobject communication.

Unfortunately, the complexity that is added when progressing from sequential to concurrent processes is also part of the deal when concurrency is introduced to object-orientation. Furthermore, one has to take into account that objects and processes differ in the sense that there is no concept of inheritance involved when traditional processes are considered.

The first part of this chapter describes some of the approaches that can be followed when combining concurrency with the object-oriented technology. As a result of concurrency, objects may affect each other due to contention and communication. Issues related to synchronisation, deadlock and race conditions are therefore discussed. This provides the background to the discussions in Chapter 4, where it is shown how these aspects are addressed in the SLOOP method.

The subsequent section is devoted to design patterns and frameworks. It provides the necessary background for Chapter 9, which demonstrates the applicability of the SLOOP method to a wide variety of typical design problems, as exemplified by the design patterns described there.

One of the distinctive features of the SLOOP method is the fact that it does not rely on the construction of a dynamic model to describe the behaviour of the system. Instead, the emphasis

is on the specification of a set of correctness properties. These properties unambiguously and precisely determine the behaviour of the system. Other representations of object interaction are described in this chapter. Finally, a brief summary is given of some of the work being done in the area of object-oriented formal methods.

In Chapter 1 it was stated that, despite the emergence of new technologies such as middleware infrastructures, it is still the responsibility of the software designer to ensure the **correctness** of the system. Other issues mentioned in Chapter 1 were **reusability, understandability and scalability**. The literature survey in this chapter covers all of these issues, as will be evident from the discussions below.

3.2 Object-orientation and concurrency

Although the introduction of concurrent features to object-oriented programming languages has been an area of research for some time now, the advent of remote execution via the World-Wide Web has highlighted the importance of this aspect of object-oriented programming. Two approaches have been followed in the process:

- New languages that are both object-oriented and concurrent have been designed.
- Concurrent constructs have been added to existing object-oriented languages.

Among the former is Orient84/K [Yoko90], while Concurrent C++ [GeRo89] and Concurrent Smalltalk [Yoko90] have evolved from their sequential counterparts.

3.2.1 Multiple threads of control

When progressing from sequential programming to concurrent programming the crux of the matter is the introduction of multiple threads of control. The obvious question to address when introducing concurrency into **object-oriented** computing is how to represent such multiple threads of control. One approach suggests the notion of active objects. The distinguishing feature of an active object is that it schedules its own operations [Meye97]. It continuously executes something similar to a "main loop".

One disadvantage of the active object approach is that a considerable amount of redefinition is often required when one active object inherits from another. It therefore adversely affects the **reusability** of these classes. Multiple inheritance is particularly problematic, since the "main loop" is an integral part of the parent class; not just another operation.

An alternative solution is to dispense with the concept of an active object, but to allow the "main loop" functionality to be defined as one of the operations of the class [Meye97]. This results in greater flexibility, since various "main loop" operations can be defined. The most appropriate one is selected for each application.

In the SLOOP method concurrency is achieved via the parallel **statements** in the parallel **operations**[1] of the classes as explained below. These statements are at a **high level of abstraction**, which has the added advantage that the target architecture can be ignored until the

[1] Since this chapter discusses issues surrounding object-orientation in general, the term "operation" is used when referring to the functions that may be applied to an object or the transformations that may be performed by an object [RBPEL91]. The implementation of an operation is called a "method" and in Smalltalk this is the terminology that is used in the class definitions [GoRo89]. Since the SLOOP notation is based on Smalltalk, SLOOP class definitions also refer to "methods". From Chapter 4 onwards, the SLOOP terminology will be used.

implementation phase is reached. The concept of inheritance also presents no problem. A brief justification of the above claims is presented next, with more detail to follow in the ensuing chapters.

Both sequential and parallel operations may be defined for a SLOOP class. The sequential operations are similar to the ones defined for classes in the conventional object-oriented methods. The parallel operations contain the parallel statements that should be executed infinitely often. By activating the appropriate parallel operations at startup, the parallel statements contained within them start executing infinitely often. **Multiple** parallel operations may be defined for a class. Furthermore, a class may **inherit** parallel operations from its ancestors and it may also **override** them. Parallel operations are activated according to the requirements of the application.

Each parallel statement may have **conditions** associated with it. If it does, the execution of the statement only changes the state of the program if its conditions evaluate to true. These conditions could represent **events**. A concurrent object therefore reacts to events via the execution of its parallel statements.

One of the goals of the SLOOP method is to provide a **unified** approach towards software design; target architectures are only considered during the implementation phase. In a SLOOP program **concurrency** is modelled by the arbitrary interleaving of the parallel statements of the program. These statements are only allocated to processes and/or processors during the implementation phase.

From the above it is evident that the SLOOP method provides **maximum flexibility** since its support of inheritance extends to its parallel operations as well. At the same time it does **not** require any **commitment** to a specific architecture before the implementation phase is reached. Concurrency support in the SLOOP method has no effect on the **reusability** of the classes. More detail regarding the structure of a SLOOP program is provided in Chapter 4. It also contains an in depth discussion of SLOOP classes, explaining the difference between sequential and parallel operations, as well as between sequential and parallel statements. Chapter 8 covers all the aspects regarding the mapping of SLOOP programs to various architectures.

3.2.2 Synchronisation

There are a number of issues related to the synchronisation of objects that need to be considered in view of the **reliability** of a software system. These issues are discussed next.

3.2.2.1 The semantics of operation invocations

The only situations in which the constituent parts of a concurrent system affect one another is when there is contention for a resource or when they have to communicate in order to pass information to one another [Bena90]. This raises the issue of synchronisation. Concurrent objects communicate via the operations that they invoke on one another. The semantics of the invocation of these operations may be either synchronous or asynchronous.

For example, the ABCL/1 concurrent object-oriented language provides three types of communication primitives, viz. past, now and future [Yoko90]. A **past** type communication primitive is **asynchronous**; the client invokes an operation on the target object and immediately continues with its execution. A **now** type communication primitive is **synchronous**; the client invokes an operation on the target object and only resumes execution once the target object has

returned a result. The **future** type communication primitive can be viewed as a **combination** of the past and now primitives. It allows the client to continue with its execution once it has invoked an operation on the target object and to synchronise with the target object at a later stage in order to obtain the result of the operation.

In order to facilitate reasoning about the correctness of a concurrent object-oriented system, it is important that the semantics should be clear when operation invocations are specified. The semantics of an operation invocation determines whether the client can assume that the postconditions of the operation hold when the client continues with its execution. Furthermore, mechanisms need to be provided to ensure that these postconditions are not invalidated until the client has completed those actions that are dependent on the outcome of the specified operation. For example, if a client may invoke an operation to remove an element from a collection only if the collection is not empty, then one should not allow the case where the result of the query issued by the client is no longer valid by the time that the operation to remove an element is performed. One way of dealing with this is to use semaphores to control access to such critical sections, but other solutions also exist. This issue is discussed in more detail in Section 3.2.2.2.

It is good practice to reflect semantic differences by notational variances, since it improves **understandability**. In the Eiffel language [Meye97] the semantics of an operation is implied by the characteristics of the client and target objects of the operation. If the two objects are in different threads of control, the keyword *separate* is used in the declaration of the target object. As a result, when the client object invokes an operation on the target object, the invocation is treated as a *separate* invocation. This means that the operation is invoked asynchronously.

For example, if target object x is declared as an attribute of the client object and object x is in a different thread of control, then the attribute declaration has the following format:

```
x: separate SOME_TYPE
```

Similarly, if the reference to the target object is passed to the client object as an argument and the target object is in a different thread of control, then the formal argument must be declared as *separate*, as shown below:

```
aaa (x: separate SOME_TYPE;  ... Other arguments ...) is
   do
      ...
   end
```

[2]

When the client object subsequently invokes an operation on the target object, the invocation will be treated as a *separate* invocation, i.e. it will be asynchronous. Synchronous invocation applies only to non-*separate* objects. Since an operation is only invoked synchronously in the Eiffel language if the target object resides in the same thread of control as the client, there is no need to take special precautions to prevent interference by other objects.

In the case of asynchronous invocations, the client and target objects are in separate threads of control, therefore there could be interference by other objects. For example, if a client object needs to check that a collection (the *separate* target object) is not empty before it removes an element, then one has to be sure that no other client object can remove elements from that

[2] A class can also be declared as *separate*, in which case each new instance of the class will be assigned to a new thread of control when the instance is created. The notation that is used to declare a *separate* class is shown below:

```
separate class X
```

collection between these two operations. The Eiffel solution to this problem is described in Section 3.2.2.2.

Great care has to be exercised to ensure that all *separate* objects are identified as such, since in some cases an object that appears to be non-*separate* could in fact be *separate* as a result of assignment or argument passing. Such objects are referred to as *traitors*. A number of rules are defined in [Meye97] in order to prevent such situations.

The SLOOP notation also allows one to indicate syntactically whether an operation invocation[3] has synchronous or asynchronous semantics. This is achieved by classifying each operation as either **sequential** or **parallel**. Invocations of sequential operations always have synchronous semantics, while invocations of parallel operations can be viewed as having asynchronous semantics. This means that when a sequential operation is invoked, the client blocks until the operation is completed, at which point the postconditions specified for that operation will hold[4]. The invocation of a parallel operation has asynchronous semantics in the sense that nothing can be assumed about the postconditions specified for the operation when control returns to the client. The postconditions will only hold if the operation is invoked infinitely often. This will be discussed in more detail in Chapter 4. The way in which interference is handled is also covered in that chapter.

Middleware products such as CORBA [OHE97] enables the designer of distributed systems to focus on the application rather on the supporting systems. It allows one to design a system without having to concern oneself with issues such as how to locate remote objects. However, it is still necessary to take into account that the operations may be executed **concurrently**. The usual concurrency issues such as mutual exclusion and deadlock prevention still need to be addressed [FGHVE96].

For example, if exclusive access to a resource is required, the CORBA Concurrency Control Service can obtain locks on behalf of transactions or threads. However, the fact that exclusive access is required, is determined by the designer.

Similarly, the decision whether to invoke an operation synchronously or asynchronously, is made by the designer. When an operation is invoked on a target object, the Object Request Broker (ORB) intercepts the message. It locates the target object, passes the necessary information to the remote site, invokes the operation on the target object, and returns the results to the client object. However, the designer has to decide whether to use **synchronous invocation** (the client invokes the operation and blocks until a result is returned), **deferred synchronous invocation** (the client continues immediately after invoking an operation and obtains the result at a later stage[5]) or **one-way invocation** (the client invokes the request and continues with its processing without obtaining a result at a later stage) [Vino97].

The synchronisation issue is particularly important from a **correctness** point of view: the program should produce the desired results and not fail due to an error condition that is purely related to concurrency issues. For example, consider the scenario as shown in Figure 3-1(a). The notation used in the diagram is from [GHJV95]. Time flows from top to bottom. Operation invocations are represented by horizontal arrows and a vertical rectangle indicates that an object is busy executing an operation.

[3] The Smalltalk (and SLOOP) terminology for invoking an operation is "sending a message".
[4] This is true if the client ensures that the preconditions hold when the operation is invoked.
[5] The client obtains the result via the `get_response` call to the ORB.

The diagram represents the object interactions if the following operations are executed by a **sequential** object-oriented program:

1. Object A invokes an operation on object B.
   In order to execute the operation object B invokes an operation on object C.
   Object B returns a result to object A once it has received a result from object C.
2. Object D invokes an operation on object C.
   In order to execute the operation object C invokes an operation on object B.
   Object C returns a result to object D once it has received a result from object B.

Note that in this specific program the order in which (1) and (2) are executed does not affect the result.

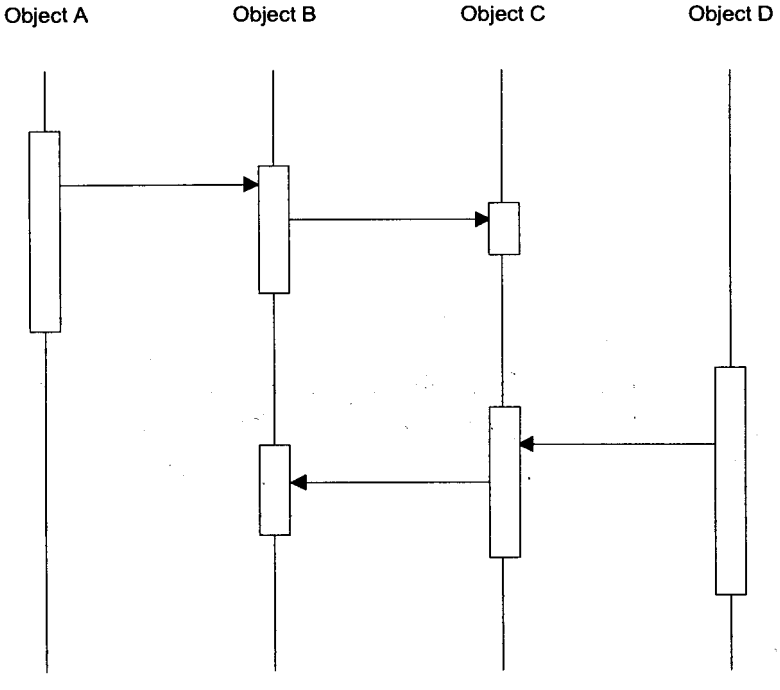

Figure 3-1(a). Synchronous operation invocations in a sequential program.

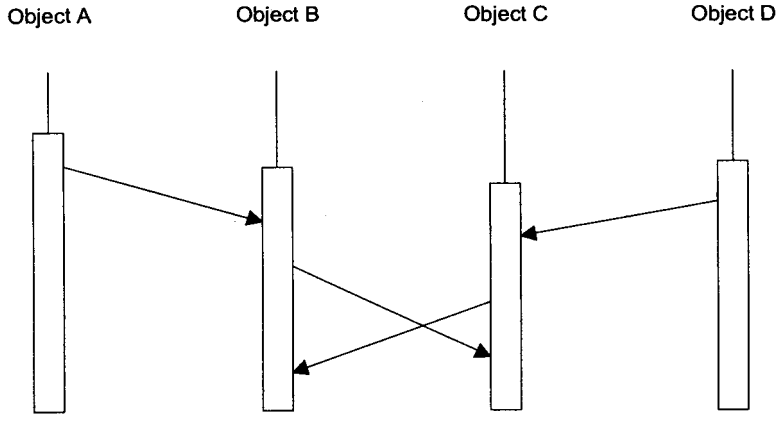

Figure 3-1(b). A scenario for deadlock during synchronous operation invocations.

The above program can easily be mapped to a distributed environment, where all the objects are in different threads of control. However, a problem arises if the following sequence of events occurs and all operations are invoked synchronously and the ORB does not support nested upcalls[6]:

1. Object A invokes an operation on object B.
2. Object D invokes an operation on object C.
3. Object B invokes an operation on object C.
4. Object C invokes an operation on object B.

A deadlock situation arises at the application level, as shown in Figure 3-1(b), since C is waiting for a response from B while B is waiting for a response from C. (The slanted arrows indicate that it may take some time for a message to arrive at another object). Eventually the supporting systems may cause a timeout, which will result in an exception for one or more of the objects, evidently not the desired outcome of the program.

In Chapter 4 it will be shown how the SLOOP notation makes provision for asynchronous and synchronous operation invocations. The SLOOP approach towards deadlock prevention is discussed briefly in that chapter and more fully in Chapter 8. At this stage it suffices to summarise it as a two-tiered approach towards synchronisation issues. At a high level of abstraction a system is designed in terms of a set of parallel statements. When reasoning about the correctness of the program, one may rely on the atomicity of each parallel statement. During the implementation phase, this atomicity must be preserved. The procedures to ensure this are not specific to the application. It is functionality that is developed once and which may then be reused by multiple applications. As suggested in Chapter 8, it could form part of the functionality provided by the SLOOP development environment.

3.2.2.2 Race conditions

As stated in the previous section, another issue which pertains to synchronisation is the reservation of objects. An accessing operation on an object might be required in order to ensure that a precondition holds before invoking the corresponding modifying operation. However, between the execution of the accessing and modifying operations the result of the accessing operation could be invalidated due to the actions of another object. Such situations, where concurrent objects may invoke accessing and modifying operations on a shared resource with the result depending on the order in which the operations are interleaved, are called race conditions. [Tane92] contains a detailed discussion of race conditions in the context of general concurrent programming.

One solution to the problem is to determine what statements constitute the critical sections of each thread of control. While a critical section is being executed, the target objects referenced in that section are allocated to the corresponding thread of control. As a result, the critical section may not be entered until those objects have been reserved for the exclusive use of that thread of control. As soon as the statements in the critical section have completed, the objects may be re-allocated.

However, the reservation procedure could result in deadlock. For example, if thread of control A reserves object X and then requests object Y, while thread of control B reserves object Y and

[6] When nested upcalls are supported, the client thread does not block while waiting for the result of a synchronous operation. Although it does not continue with its execution, the original client may service operation invocations from objects in other threads, including from the target object.

requests object X, deadlock arises since each thread of control is requesting a resource held by the other and neither will release the currently held resource until the other resource has been obtained[7].

In order to ensure that the reservation of multiple objects by multiple threads of control does not result in deadlock, an (arbitrary) order can be assigned to the relevant objects [Tane92]. Since objects may only be requested and allocated in this order, the above scenario cannot occur. For example, if objects are reserved in lexicographical order, thread of control B cannot reserve object Y before it reserves object X. It will request object X first. If the latter has already been allocated to thread of control A, then thread of control B will just wait. Eventually object Y will also be allocated to thread of control A. Once all the objects have been released by thread of control A, object X will be allocated to thread of control B, after which it may request object Y.

If a programming language does not offer more advanced concurrent facilities, semaphores may be used to control access into critical sections. One example of language support for controlling access into critical sections is the way in which argument passing is interpreted as an object reservation mechanism in the Eiffel language [Meye97]. When successive operations on a target object require exclusive access to the latter they should be enclosed in an operation of the **client** object. The target object must be declared as a *separate* formal argument of the specified client operation, which indicates to the compiler that exclusive access to the target object is required for the duration of the client operation.

The **atomicity** of the SLOOP parallel statement, together with its **expressive power**, provides the software designer with the capability to control access to critical sections in SLOOP programs. This topic is revisited in Chapter 4.

3.2.2.3 Synchronisation constraints

As stated earlier, many object-oriented programming languages were initially designed for sequential programming. One of the options that has been investigated as a way of incorporating concurrency into object-oriented methods, is the concept of path expressions. The idea is to leave the definitions of the classes and their operations unchanged, but to add path expressions specifying which operations are allowed in what object states. The path expressions therefore determine how objects are allowed to synchronise with each other. These synchronisation constraints are **external** to the class specifications. A fair amount of redefinition might therefore be required during inheritance [Meye97]. For example, if a new operation is added in a subclass, then all the path expressions would have to be updated where this new operation is allowed.

The order in which operations may be invoked on a SLOOP class or object is fully specified by the preconditions of the correctness properties of that class. This is similar to the way in which synchronisation constraints are expressed in Eiffel [Meye97]. Since the correctness properties form part of the specification of a class and its methods, inheritance does not present any problems. For example, if a subclass contains a new method, then one or more correctness properties are specified as part of the method definition. These properties provide the necessary information regarding the conditions under which the new method may be used. In Chapter 7, Section 7.3.1.4, it will be shown how correctness properties represent synchronisation constraints in the SLOOP method.

[7] The conditions that need to hold for deadlock to occur, will be described in more detail in Chapter 4.

3.3 Design patterns and frameworks

Chapter 9 provides examples of how various design patterns can be incorporated into a system that was designed using the SLOOP method. Special consideration is given to the impact of the SLOOP computational model on the application of these design patterns. In order to place it in context, a brief introduction to the relevant aspects of design patterns and frameworks is presented next.

3.3.1 Categories of patterns

One of the difficult aspects of designing software is to achieve the goal of reusability. Although this aspect of software engineering receives a great deal of attention in object-oriented techniques, the level of abstraction when dealing with **classes** is fairly low. More recently the emphasis has shifted towards **design** reuse. As a result the concept of a design pattern has evolved. A **pattern** describes a recurring problem and the crux of a solution to it. The pattern is used as a template, with each implementation which reuses the pattern potentially different from every other implementation of the pattern. Design patterns are patterns at a certain level of abstraction.

In [BMRSS96] three different categories of patterns are identified, based on three different levels of abstraction:

At the highest level is the **architectural pattern.** It describes the structure of a software system in terms of all its subsystems and the interactions between them. An architectural pattern can be used to build a framework, which would then include the relevant subclasses. One example of an architectural pattern is the Model/View/Controller (MVC) system used to build user interfaces. Another example is the Layers architectural pattern used in networking systems.

Design patterns are medium-scale patterns. They are defined as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [GHJV95]. These patterns neither describe simple classes such as linked lists nor complex domain-specific applications.

Idioms are low level patterns that are language-specific [BMRSS96]. For example, the reference counter idiom is used to keep track of dynamically allocated resources in C++ programs, whereas the garbage collection idiom performs a similar function in Smalltalk.

Other concepts related to patterns are [Vilj95]:
- paradigm - a style of work that pervades the whole system,
- principle - a design rule that always holds,
- heuristics - a design aid that will not necessarily result in the best options being selected.

The following definition of a pattern is from [BMRSS96]: "A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate."

As Viljaama points out, a design pattern is "more than a recurring solution" [Vilj95]. It is a specific way of describing the solution, including the context in which it is applicable and the "forces" or constraints affecting the solution.

The term "pattern language" as used by the architect Christopher Alexander refers to a set of patterns, where each pattern is used to solve a particular kind of problem [John92]. It is not a formal language, such as a context-free language, but structured prose. Alexander refers to it as a language, because in the same way as one could construct any sentence using the words of a language, one could design any building by applying his patterns one after the other. The same does not apply to design patterns. It is not likely that it would be possible to create a set of design patterns so complete that one could claim that any program could be constructed by merely applying those design patterns [GHJV95].

3.3.2 Advantages of using design patterns

Using design patterns has the following advantages:

- It provides a common vocabulary, which means that once designers are familiar with the designs represented by various design pattern names, they can communicate by referring to these design pattern names rather than having to describe each design in detail [BMRSS96].
- Design patterns provide a vehicle for documenting well-proven design experience [BMRSS96].
- Design patterns can be used to construct the building blocks for complex systems. They can therefore be regarded as a mechanism to manage complexity [BMRSS96].
- Systems are more likely to exhibit non-functional properties such as maintainability, modifiability, reliability, and interoperability if they are constructed using design patterns that have these properties [BMRSS96].
- There are many approaches towards identifying objects. One of these is to keep the solution space as close as possible to the problem space, i.e. the objects that are identified should have counterparts in the real world. However, quite often other abstractions emerge when a system is designed with reusability and flexibility in mind. Design patterns help to identify such abstractions, e.g. the Strategy pattern provides a common interface which allows one to exchange one algorithm for another [GHJV95].
- Design patterns encourage programming to an interface, rather than to an implementation. They specify the key elements that should be present in the interface and in some cases, such as the Decorator and Proxy patterns, they also specify the relationships between the interfaces of classes (the interfaces of the decorated and proxied objects have to be identical to those of the Decorator and Proxy objects) [GHJV95].
- Design patterns promote the usage of object composition as a reuse technique, while providing the guidelines to do it in a controlled way [GHJV95]. More details regarding this aspect are given next.

Two of the most common reuse mechanisms are object composition and class inheritance. The former is often referred to as black-box reuse, whereas the latter can be considered white-box reuse [GHJV95]. Class inheritance often requires knowledge of the implementation of the parent classes. Object composition only requires the interfaces to be known. It therefore promotes encapsulation.

It is often possible to achieve as much flexibility and extensibility with object composition as with class inheritance. This is done by delegating tasks to other objects. For example, instead of making a Window class a subclass of Rectangle if the window is rectangular, the Window class delegates the relevant operations (such as calculating the area) to the Rectangle class

[GHJV95]. The Window class has an instance variable referring to the Rectangle instance handling the forwarded operations.

The software architecture of a system can become very complex if delegation is not used in a controlled way such as in design patterns.

The concept of design patterns is relevant to the SLOOP method, not only because it is used to illustrate that the SLOOP method can be applied to various types of design problems, but also because the incorporation of design patterns into a SLOOP design is an **explicit action** in the SLOOP method. It is performed in order to take advantage of the benefits of using design patterns as listed above. Details of the various steps that comprise the SLOOP method are given in Chapter 4.

3.3.3 Design patterns and formalisation

The information to be conveyed in a design pattern is documented in a consistent way using a template. Several templates exist, but the basic components are as follows [BMRSS96]:

- Context: design situation giving rise to a design problem
- Problem: set of forces repeatedly arising in the context
- Solution: configuration to balance the forces:
  - structure with components and relationships
  - run-time behaviour.

The term *forces* is used to denote all the aspects that need to be taken into account when devising a solution to the problem, viz. the functionality that is required, the constraints that apply and the properties that are desirable (e.g. reliability and maintainability) [BMRSS96].

Current design pattern descriptions are informal. The possibility of formalising these descriptions is being investigated, since that would result in more precise pattern descriptions and pattern tools could then be developed more easily. One example of how the collaboration among the participants in a design pattern can be described formally is given in [Mikk98]. However, another point of view is that formalisation would make the problem and solution descriptions of the pattern harder to understand [BMRSS96]. It is also difficult to envisage a formalism that could describe the benefits and liabilities as well as the implementation guidelines of a design pattern. In [GHJV95], Gamma et al. focus on exploring the pattern space rather than formalising design patterns.

The purpose of this research is not to formalise the description of design patterns. However, when a design pattern is incorporated into a SLOOP design, the behaviour of the collaborating objects is described by the **correctness properties** of these objects, which makes the SLOOP descriptions **precise** and **unambiguous**.

3.3.4 Frameworks

Similarly to a design pattern, a framework comprises a set of co-operating classes. In both cases there are static and dynamic aspects involved. However, where a design pattern presents a solution to a general design problem, a framework applies to a specific class of software, such as user interfaces, commercial data processing systems or telecommunications. Design patterns are smaller architectural elements than frameworks. A framework usually contains one or more design patterns, but not vice versa. A design pattern is only a template for a solution, i.e. each time the design pattern is reused, it needs to be implemented, whereas a framework is a "semi-complete" application which needs to be **instantiated**.

A framework therefore defines the architecture of a software system or subsystem by describing its constituent classes and the relationships between them. It also specifies the thread of control. The framework is instantiated by applying object composition and subclassing techniques.

Some parts of a framework are designed to remain unchanged in any instantiation, so-called "frozen spots" [BMRSS96, Pree94]. They dictate the architecture of the system and define the basic elements and the relationships between them. The "hot spots" [BMRSS96, Pree94] are those places where adaptations can be made to suit the requirements of individual applications.

In the case of code reuse, the user has to design the main program around the building blocks being provided. Frameworks, on the other hand, dictate the architecture of the main program and the user has to provide the building blocks. Similar applications will therefore have similar structures if the same framework is used. That has the advantage of improved **maintainability**.

Another advantage of frameworks is that it captures the essence of successful patterns, architectures and programming mechanisms.

Again, the SLOOP method takes advantage of the benefits of using frameworks. The design phase of the SLOOP method has **explicit steps** to search the repository of reusable artifacts for framework(s) that are applicable to the problem at hand. It uses the correctness properties resulting from the analysis phase to identify suitable framework(s).

3.4 Specifying object interaction

The specification of object interaction is a very important aspect of the SLOOP method. This section provides some background to the topic.

The design of a system involves both static and dynamic aspects. The structure, i.e. the various classes and their relationships, represents the static part of the design. The object interactions constitute the dynamic part of the design. There are many approaches towards specifying interactions between objects.

The dynamic model of the Object Modeling Technique (OMT) [RBPEL91], consists of a **set** of state diagrams, one for each class with important dynamic behaviour. All the possible states of an object are captured in its state diagram. Events may cause transitions from one state to another. An event is defined as "an individual stimulus from one object to another" [RBPEL91]. The state diagrams are related to one another via shared events.

Other examples of graphical representations of object interaction are the interaction diagrams shown in [GHJV95] and the Object Message Sequencing Charts (OMSCs) in [BMRSS96]. These diagrams describe **specific scenarios**, as opposed to specifying **all possible behaviours** of the objects involved. Also, the focus is on **object interaction**, rather than on all the possible state transitions of **individual** objects. The generic form of the sequence diagrams of the Unified Modeling Language (UML) [RSC97] allows one to specify all possible execution sequences, since it provides representations for loops and branches. The instance form of the UML sequence diagram is used to describe specific scenarios. Clearly the facility to describe the behaviour of objects generically is a desirable feature of an object interaction notation, since it enables one to summarise many different scenarios in a single diagram or description.

Traditionally, the emphasis is on the static structure of the architecture in object-oriented design. In contrast, Helm et al. advocate **interaction-oriented** design in [HHG90]. They claim

that it promotes the early identification, abstraction and reuse of patterns of behaviour in programs. The behavioural dependencies are captured in **contracts**. A contract identifies each object that participates in the provision of the functionality described by the contract. It describes the contractual obligations of each object in terms of type and causal obligations. The type obligations specify the external interface of the object as required by the contract. The causal obligations dictate the order and results of the actions performed by the object.

The contracts are defined independently of the class specifications. A conformance declaration is used to specify how a class meets the contractual obligations of a contract. It describes how the obligations are distributed over itself and its subclasses. A class may participate in many contracts, therefore many conformance declarations may be associated with a class.

Refinement and composition techniques are used to specify complex contracts in terms of simpler ones.

The design of an application includes a number of steps:
- the specification of the behaviour using contracts,
- the conformance declarations of the participating classes and
- the instantiation of the contracts.

Constructs are provided to instantiate a contract from within an application or class. When a contract is instantiated, it ensures that its specified initial obligations are met. Thereafter the causal obligations of its participants dictate its behaviour.

A different type of contract is described in [Meye97]. In the Eiffel programming language invariants are specified for each class and pre- and postconditions govern the execution of each operation. These assertions may be checked during run-time. Meyer refers to this as the "design by contract" approach [Meye97].

Essentially, design patterns are about object collaboration. However, it is at a very high level of abstraction. They do not contain any application-specific information. Contracts can be used to describe the interactions between the participating objects in an application and are therefore not as generic as design patterns.

Lajoie and Keller suggest that yet another level of abstraction is required when designing frameworks [LaKe94]. They recommend that a framework should contain design patterns, contracts and **motifs**. The latter is used to provide an informal description of the purpose of the framework as well as how to use it. Motifs should also describe the purpose and usage of framework application examples. The motif should capture those issues that are important in order to **preserve the design during modifications**. As a result they often refer to design patterns and/or contracts. A motif template to guide the designer is provided in [LaKe94].

In the SLOOP method **object collaboration** is specified primarily via the **correctness properties of the classes**. These properties specify the **required behaviour** of the classes and their operations. The SLOOP parallel statements specify how this behaviour is **realised**. Details of this aspect of the method are provided in Chapters 5, 6 and 7. The specification of the correctness properties could therefore be viewed as a description of **what** needs to be complied with, whereas the SLOOP parallel statements can be seen as a specification of **how** this is achieved.

3.5 Object-oriented formal methods

The advantages of combining object-orientation concepts with formal methods have been noted by many researchers. Ruiz-Delgado et al. list the excellent **structuring capabilities** of object-orientation as one of the main benefits [RPS95]. It could also make formal methods **easier to use**.

A survey of object-oriented formal specification languages that have been developed fairly recently is presented in [RPS95]. There are two main development approaches: some researchers focus on adding object-oriented concepts to existing formal specification languages while others develop new languages that contain these features from inception.

Examples of the former are Object-Z, OOZE (Object-Oriented Z Environment), Z++, LOOPN (Language for Object-Oriented Petri Nets) and CO-OPN (Concurrent Object-Oriented Petri Nets) [RPS95]. Amongst the new languages that are not based on existing ones are CSML (Conceptual Model Specification Language), MONDEL and ENVISAGER [RPS95].

As stated in Chapter 1, the SLOOP method follows a **"lightweight"** approach towards software development. The aim is not to come up with a new formal method. Instead, the goal is to provide a design approach that is likely to yield more reliable systems, due to its more rigorous basis. The **object-oriented** nature of the method is particularly important in order to address the **scalability** problem of such a rigorous approach. The way in which the method is applied will be illustrated via a case study from Chapter 5 onwards.

3.6 Summary

This chapter served as an introduction to the object-oriented aspects relevant to the SLOOP method. For each aspect some of the existing approaches were described, followed by a brief description of the way in which the issue is addressed in the SLOOP method.

The first topic was concerned with the issues resulting from the combination of **concurrency** with **object-oriented** concepts. It emerged that a concurrent object-oriented method should provide a **convenient mechanism** to represent the **concurrent behaviour** of the objects, without sacrificing the **reusability** and **extensibility** benefits that are achieved via inheritance. A brief introduction to the way in which the SLOOP method ensures this was given.

The notation should allow one to indicate semantic differences via syntactical variances, since it aids **understandability**. In particular, it should be possible to distinguish between synchronous and asynchronous operation invocations. This makes it easier to reason about the **correctness** of the program, since the semantics of each statement can be deduced from the syntax.

It is also necessary to be able to indicate atomicity in order to prevent interference resulting from **race conditions**. Another desirable feature is a convenient mechanism to express **synchronisation constraints**. Again an introductory description was given of the way in which the SLOOP method addresses these **reliability** issues.

The second part of this chapter was devoted to a discussion of **design patterns** and **frameworks**. The benefits of this technology were described, because the **reuse** of design patterns and frameworks forms an **integral part** of the SLOOP method.

Since the emphasis in this thesis is on **precise** and **unambiguous specifications**, some of the current views on the formalisation of design patterns were presented. It was noted that when design patterns are incorporated into a SLOOP design, then the behaviour of the collaborating objects is specified via the correctness properties.

The representation of **object interactions** was the third topic covered in this chapter. **Expressive power** emerged as an important requirement of the notation, i.e. it should be possible to represent interactions generically rather than via scenarios only. In the SLOOP method the **correctness properties** define the behaviour of the system. This description of **what** the behaviour should be is further enhanced by a description of **how** this is achieved (via the parallel statements). The SLOOP representation of object interactions is generic.

The final topic that was discussed was concerned with **object-oriented formal methods**. It was noted that object-orientation could make formal methods easier to use. The inherent structuring capabilities of object-orientation is another reason to combine the two technologies. Although the SLOOP method is not formal, one of its goals is to promote a more rigorous approach towards software development. One of the reasons for including object-orientation concepts in the SLOOP method is to address the **scalability** problem usually associated with more formal approaches.

This concludes the introductory part of the thesis. The next chapter presents an overview of the SLOOP method. Many of the design choices described in the remaining chapters are based on the results of the literature study that was discussed in this and the preceding chapters. These design choices will be highlighted as they are encountered.

CHAPTER 4

THE SLOOP METHOD

4.1 Introduction

This chapter introduces the SLOOP method. In Chapter 1 the desired features of the method were described. To recapitulate:

- It should assist the software designer in producing reliable and functionally correct software systems.
- It should be suitable for the development of the whole spectrum of small- to large-scale systems.
- The method should be usable by software engineers who are not necessarily proficient in the mathematics required by formal methods.
- It should be applicable to all types of software architectures, from sequential to concurrent and distributed.
- It should support mechanisms that facilitate reuse at multiple levels of abstraction.
- The transition between the various software development phases should be seamless (a relatively simple mapping to an executable language would facilitate rapid prototyping).
- The target implementation language should be well-known and generally available. In order to facilitate a seamless transition between the specification and implementation of a system, the specification language needs to resemble the implementation language.
- One of the purposes of developing the SLOOP method is to evaluate the merit of a "lightweight" formal method in terms of the benefits that it provides. Since it is an experimental system, it should minimise developmental resources.

Chapters 2 and 3 provided further background to the issues addressed by the SLOOP method. All of these are discussed in this chapter:

- the "constructive approach" towards software development,
- the computational model,
- the representation of the correctness properties,
- object-orientation and concurrency,
- synchronisation issues,
- design patterns and frameworks and
- the specification of object collaboration.

The purpose of this chapter is to provide an overview of all the aspects of the SLOOP method. Some topics are vast, in which case a summary of the main concepts are given in this chapter, while the detail is covered in subsequent chapters. For example, the identification of correctness properties is described in Chapters 5 and 6, while informal correctness reasoning is discussed in Chapter 7. This chapter only indicates where these aspects of the SLOOP method fit into the big picture. The design

choices that were made while devising the SLOOP method will be highlighted as they are encountered.

The remainder of this chapter is organised as follows: First of all, an overview of the method is given. Since the SLOOP program captures the essence of the method, the major part of this chapter is devoted to describing the structure of a SLOOP program. The latter part of Chapter 4 introduces analysis, design and implementation within the SLOOP context. Subsequent chapters elaborate on these aspects of the method.

4.2 Overview of the SLOOP method

The SLOOP method provides more than just a notation; it guides the user to approach **object-oriented** software development from a specific angle. This is as a result of two of its underlying characteristics, i.e.

- the reduced role of control flow in program design (due to its computational model) and
- the fact that correctness reasoning forms an integral part of all the software development phases.

Reuse, which is the third cornerstone of the method, is included as a matter of course when object-oriented design is under consideration. However, in this case reuse is extended to include the reuse of the correctness properties as well as the reuse of mappings.

The SLOOP method encourages a systematic and disciplined approach towards software development spanning the entire development lifecycle. The specification and the preservation of the correctness properties **steer** the software development process.

4.2.1 The SLOOP computational model

The SLOOP computational model affects the system design in the sense that one has to keep in mind that the core of the SLOOP program is a set of parallel statements[1] that execute infinitely often and in any order. Each parallel statement executes **atomically**.

Instead of making correctness assertions based on where the locus of control is in each of the constituent concurrent objects of the system, they are made without taking program location counters into account. The concept of control flow is therefore irrelevant. This characteristic of a SLOOP program makes it possible to classify it as a Single Location Program (SLP). In [GePn89] it was shown that the concept of control flow is irrelevant in this class of programs.

This approach is particularly suited to the object-oriented paradigm, because the emphasis in object-oriented design is not on a control flow driven from a main procedure as in functional structured design. Instead, a system comprises a set of objects[2] that send messages to one another, resulting in the execution of the corresponding methods. The SLOOP parallel statements provide a way to represent object **interactions** with the focus on their properties instead of on their flow of control.

Note that there could be restrictions on the order in which operations should be invoked. Such dependencies are reflected by the correctness properties of the class and/or its operations, as was discussed in Section 3.2.2.3.

---

[1] A SLOOP parallel statement is similar to a UNITY multiple assignment statement. It is described in detail later in this chapter.

[2] A class "describes the implementation of a set of objects that all represent the same kind of system component" [GoRo89]. An object is an instance of a class [GoRo89].

The SLOOP method could be viewed as a design discipline: some constraints are imposed on the design process in order to arrive at a design of higher quality. Thus, it is an aid in achieving the ultimate aim of producing **reliable** and **functionally correct** software. The designer has to take care to base the design on the SLOOP computational model, i.e. *the statements have to be designed in such a way that they can be executed infinitely often in any order and that such an execution would result in the desired outcome. The design of such a set of statements is made easier if the point of departure is a set of correctness properties describing the behaviour of the system.* No consideration is given to target architectures during the analysis and design phases.

When the above approach is followed, the resulting program is more amenable to correctness reasoning, since it is not necessary to take location counters into account. This can be compared with a design discipline that discourages the use of the goto statement, because it also has the effect of simplifying correctness reasoning.

4.2.2 The main components of the method

The SLOOP method provides the following:

- a guideline for the steps to be followed during the **analysis**, **design** and **implementation** phases, with the emphasis on the correctness properties of the system,
- provision for a repository of **reusable** classes, patterns, frameworks and mappings,
- the **SLOOP notation** which enables the designer to model the behaviour of the system precisely and unambiguously,
- a basis for **reasoning informally about the correctness properties** of the system,
- **mapping heuristics** which facilitate the mapping of a SLOOP program to an executable Smalltalk program.

The executable program may be the final product, or it may be used for rapid prototyping or to generate traces. The reflection facilities of Smalltalk can be used to perform selective assertion checking and to generate traces.

An overview of the above features is given next.

As depicted in Figure 4-1, the method guides the user to arrive at a design that can be reasoned about, starting from a (possibly amorphous) set of problem domain objects. During the analysis phase the problem domain is studied. This results in a **class diagram** representing the **static structure** of the problem domain. The **behavioural** characteristics are captured by describing the correctness properties of the system informally.

The design phase elaborates on these representations. Quite often this results in the inclusion of classes that represent concepts that have no counterparts in the real world. Some of these are low level classes such as arrays, while others may be at a high level of abstraction introduced specifically to make the design more reusable.

Some or all of the software artifacts identified during the analysis and design phases may be selected from a repository of reusable classes, patterns and frameworks.

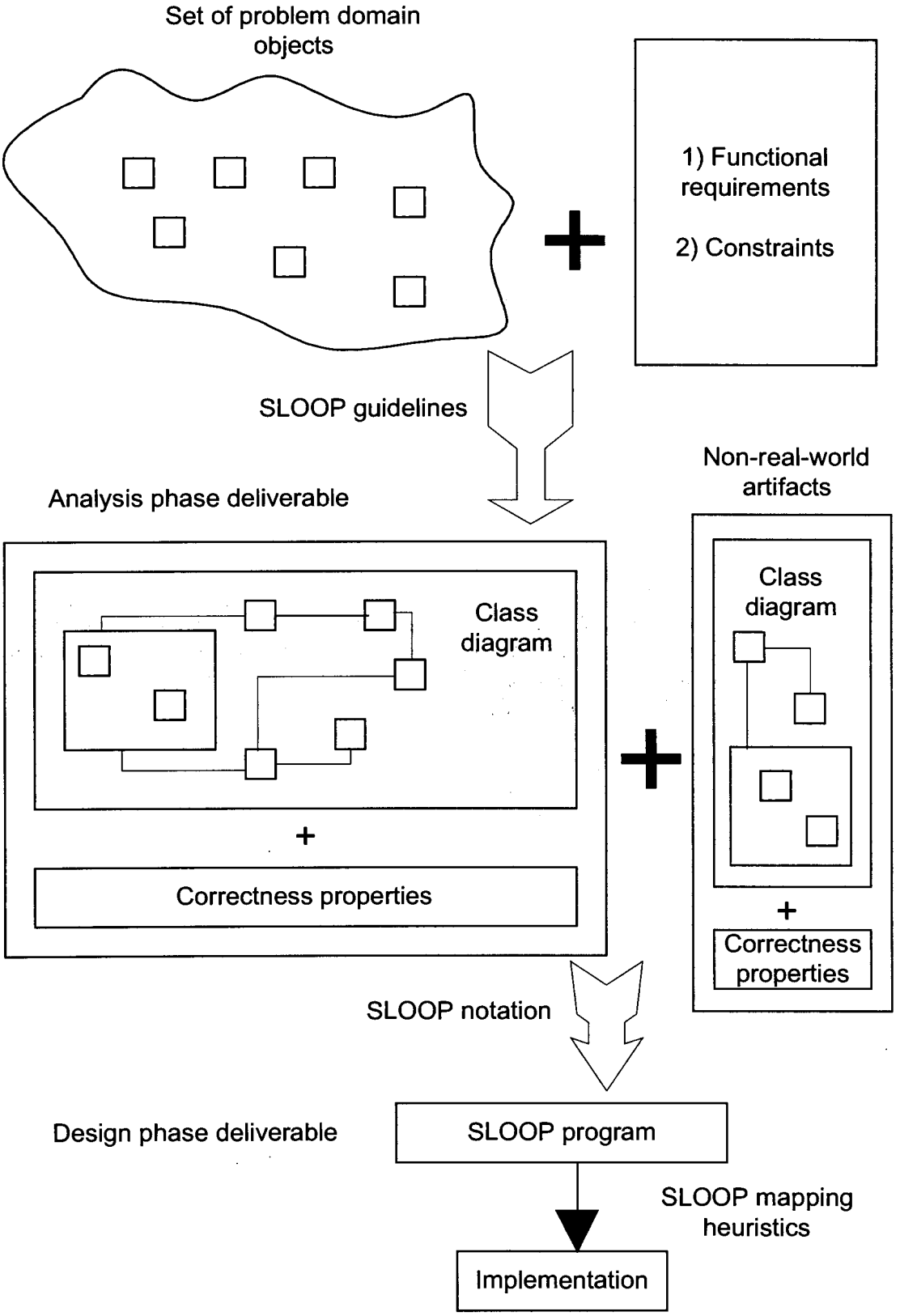

Figure 4-1. Overview of the SLOOP method.

The result of the design phase is a SLOOP program which can be mapped onto an executable Smalltalk program. As stated in Chapter 1, a SLOOP program also contains specification aspects, notably the correctness properties that should be satisfied during execution. These correctness properties are specified more rigorously during the design phase. The informal correctness arguments that are used to reason about these properties may be stored in the repository along with the associated reusable classes. Having the capability to specify properties **within** the program encourages a **design based on correctness reasoning**. It also serves as a way of **documenting** the **behaviour** of the program.

If required, some of these **properties can be checked** during run-time using the **reflective** capabilities of Smalltalk. A pragmatic approach towards assertion checking should be followed. The SLOOP notation for specifying properties contains constructs that allow for the specification of first-order predicates. These constructs are not available in Smalltalk and when mapped to Smalltalk enumeration messages, the corresponding assertions can be very expensive to check.

Note that the above assertion checking feature is not purported to be an automated validation method. Most automated validation methods require a reachability analysis[3] [Holz91]. Assertion checking as used here has the purpose of ensuring that preconditions are met, that class invariants hold before an operation is executed and that violations of postconditions and class invariants after the execution of an operation are detected. Appropriate action can be taken if a violation is detected during a specific execution. During the mapping of a SLOOP program to an executable Smalltalk program, the designer therefore decides which assertions need to be checked in this way.

The SLOOP notation bears similarities with both UNITY and Smalltalk. The UNITY-like nature of the SLOOP parallel statements simplifies the correctness reasoning, since there is no need to take location counters into account. The Smalltalk-like aspects simplify mapping to an executable program[4]. This facilitates **rapid prototyping** during the design phase. The derived Smalltalk program can also be used to generate traces, once again using the reflective facilities of Smalltalk.

The Smalltalk terminology [GoRo89] is adopted in the remainder of this thesis when describing object-oriented concepts. For example, when an operation is invoked, it is said that a *message* is sent to the *receiver*, the latter being the target object. If the message is sent to *self*, the object sending the message is also the *receiver*. A message consists of a *selector* and zero or more *arguments*. A *message expression* is the combination of the *receiver* and the *message*. Since a message expression returns an object, a message argument may itself also be a message expression.

The implementation of an operation is called a *method*. The *selector* that identifies the method and the *pseudo-variables* that represent the arguments[5] are collectively referred to as the *message pattern*. Note that the value of a *pseudo-variable* cannot be changed via an assignment expression within the method implementation [GoRo89]. If a message is sent to *super*, the search for the corresponding method starts in the superclass of the object sending the message.

---

[3] The reachability analysis does not necessarily have to be exhaustive. Due to the "state space explosion problem", many controlled partial search techniques have been developed that produce satisfactory results, i.e. the major functionality is tested and the probability of finding any given error is higher than the coverage, where coverage is defined as the number of reachable states analysed divided by the total number of reachable states [Holz91].

[4] As noted in Chapter 1, this similarity with Smalltalk does not preclude mappings to other object-oriented languages. A mapping to Java is one possibility, because Smalltalk to Java translation can be done quite successfully, as has been reported in [Enko98].

[5] Pseudo-variables therefore correspond to "formal parameters" in imperative programming languages.

The attributes of a class are represented by *class variables* and *instance variables*. The contents of a class variable is available to all instances of the class. The scope of an instance variable is local to a class instance. As in Smalltalk, each entity in the solution domain is viewed as an object. No static type checking is performed. Temporary variables (variables that are local to a method) are not declared in SLOOP. Since there is no static type checking, it was decided that there was no need to declare any variables that did not represent attributes of the class.

Smalltalk differentiates between *class methods* and *instance methods*, the former being the methods defined for the metaclass[6] of the class and the latter being the methods that are executed by instances of the class. The SLOOP notation makes the same distinction.

One of the motivating factors for allowing Smalltalk message expressions in the SLOOP notation is the fact that it provides instant access to an **extensive class library**. However, the use of this library does have one restriction: messages that are related to the Smalltalk-80 support for multiple independent processes [GoRo89] may not be used[7]. This is because there is no concept of a process in a SLOOP program.

For example, in the Smalltalk-80 context, when the message `wait` is sent to an instance of the class `Delay`, it results in the active process being suspended for a specified period. There is no notion of a process performing a blocking wait in a SLOOP program. The case study that is described in the remainder of this thesis illustrates how timers can be implemented in the SLOOP idiom. (Details can be found in Sections B.11 and B.12 in Appendix B.) The above restriction ensures that no synchronisation implicit to the Smalltalk language is carried over to SLOOP programs. Note that this restriction does not apply to the executable Smalltalk programs that are created during the implementation phase, when the SLOOP program is mapped to specific target architectures. This aspect is described in more detail in Chapter 8.

4.2.3 The crux of the method

The SLOOP parallel statements capture the essence of the SLOOP approach, since they represent the embodiment of the computational model. The difference between sequential and parallel **methods**, as well as between sequential and parallel **statements** is described next in order to provide the necessary background for the ensuing discussions.

In addition to the usual classification of methods into class and instance methods, SLOOP methods are also categorised based on the type of statements that they may contain. Two different categories are defined, viz. *sequential* and *parallel*. A method is therefore a class **or** instance method **and** it is a sequential **or** parallel method.

The following definitions apply:

Sequential method	It contains sequential statements only. The order in which the statements appear is significant. A sequential method is similar to

[6] In Smalltalk-80 all system components are viewed as objects. Classes must therefore themselves be instances of a class. "A class whose instances are themselves classes, is called a metaclass" [GoRo89].

[7] The Smalltalk-80 system makes provision for concurrency via the concept of multiple independent processes [GoRo89]. Each process is a sequence of actions described by message expressions. The processes are scheduled by an instance of the class ProcessScheduler. A process may be suspended if it is sent the message *suspend*. Another message that causes the suspension of a process is *wait*. It is used to implement semaphores and also to synchronise with the real-time clock.

	a method in a conventional object-oriented language. The statements within a sequential method are subject to the same rules of flow of control as the statements of a method in a conventional object-oriented language. A **sequential method** is always executed **atomically**.
Parallel method	It contains parallel statements only. The order in which the statements appear is not significant. There is no concept of flow of control. The unit of **atomicity** is the **parallel statement**, not the parallel method.
Sequential statement	Each sequential statement within a sequential method is executed whenever the encapsulating method is invoked, provided the flow of control reaches that statement. A sequential statement may contain other sequential method invocations, but not any parallel method invocations. This restriction is necessary since a sequential method needs to satisfy a total correctness property. The difference in how correctness properties are interpreted for sequential and parallel methods is decribed in more detail in Section 4.3.4.3.
Parallel statement	Only one[8] parallel statement is executed whenever the encapsulating method is invoked. Any one of its statements may be executed, provided the SLOOP fairness requirement is satisfied. The latter specifies that each parallel statement must be executed infinitely often. The method therefore has to be invoked infinitely often. Such infinitely often invocations have to result in the infinitely often execution of **each** parallel statement contained within the method[9]. A parallel statement may contain other parallel or sequential method invocations.

Figure 4-2 contains some scenarios illustrating the above concepts. In the diagram the steps represented by *A* are executed as a single atomic unit. Thus, when parallel method PM-1 is invoked and statement ps-1.1 is executed, sequential method SM-1 is invoked, resulting in the execution of sequential statements ss-1.1 and ss-1.2. Thereafter sequential method SM-2 is invoked, resulting in the execution of sequential statements ss-2.1 and ss-2.2. Together these steps form a single atomic unit.

When parallel method PM-1 is invoked again, statement ps-1.2 might be selected for execution. In that case sequential method SM-3 is invoked, resulting in the execution of sequential statements ss-3.1 and ss-3.2. The last sequence, ending in the '*X*' symbol, is not allowed, since a sequential statement may not invoke a parallel method.

In Chapter 2, Section 2.3.1, it was stated that concurrency can be modelled by the **arbitrary interleaving** of **atomic** instructions from different processes. Thus, at any moment only one atomic instruction from one of the processes is executing. This instruction executes to completion before the next atomic instruction of the same or a different process is selected arbitrarily. In the SLOOP context, the atomic instruction is the parallel statement. In the above example, each of parallel statements ps-1.1 and ps1.2 executes as an atomic unit. They may be executed in any order, but not

[8] The rationale for this restriction is given later in this section.
[9] The way in which the infinitely often execution of each parallel statement can be achieved during the implementation phase is discussed briefly in Section 4.4.3.3 and in more detail in Chapter 8.

simultaneously. In a SLOOP program concurrency is therefore modelled by the arbitrary interleaving of parallel statements.

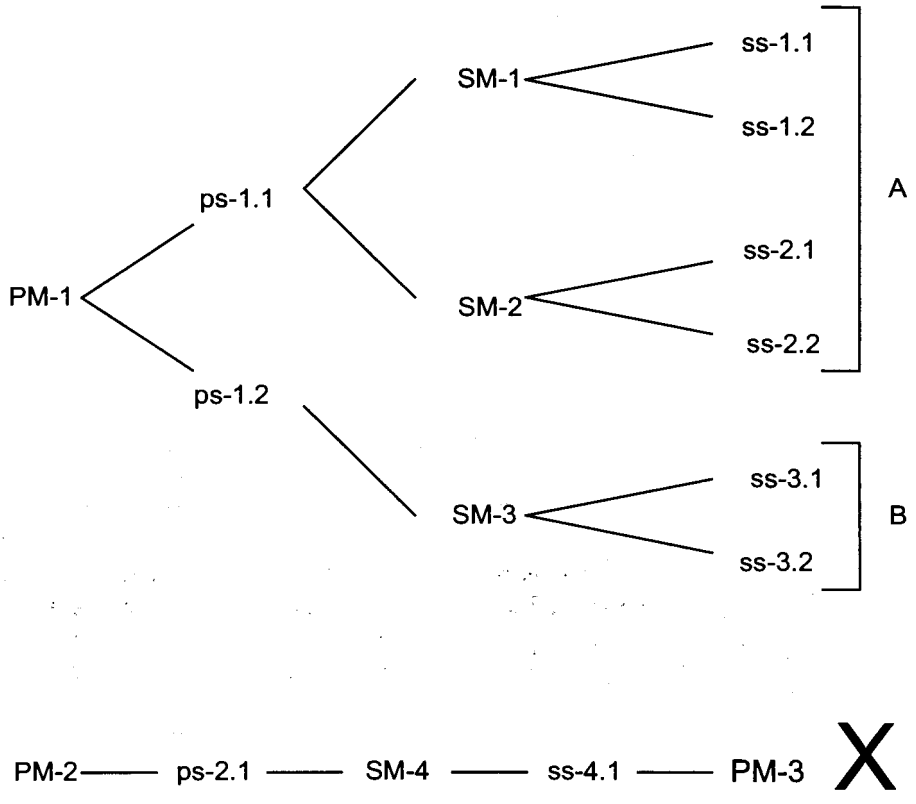

Figure 4-2. Scenarios representing atomic executions.

During the implementation phase the parallel statements are assigned to processor(s). The complete spectrum of configurations can be used without affecting the correctness of the design: from assigning all the parallel statements of the program to a single processor to assigning each parallel statement to a different processor.

The **rationale** for defining the above method categories is presented next.

As explained in Chapter 2, UNITY programs are based on the concept of multiple assignment statements that execute infinitely often. These are **assignment** statements, i.e. they contain assignments of **values** to **variables**. The result of the execution of a **terminating** function may be used as the value that is assigned to a variable. The function has to be terminating in order to satisfy the fairness requirement.

The SLOOP method explores the use of these concepts in the object-oriented paradigm. Thus, instead of restricting the statements to multiple assignment statements involving variables and values, the statements are formulated in the object-oriented mould. The emphasis is on objects and the messages sent to these objects. Although an assignment statement is a valid SLOOP statement, the SLOOP method takes advantage of the higher level of **abstraction** provided by message expressions, as illustrated in the program excerpt below.

The purpose of the two parallel statements in the example is to receive objects from a producer, transform them and subsequently buffer them until they can be passed to a consumer on a First In First Out basis, while ensuring that a record is kept of the maximum length ever reached by `bufferedElements`, the FIFO queue containing the buffered objects. It is a bounded buffer, hence the requirement for the `maximumAllowedLength` attribute. The `maximumRecordedLength` attribute is used for statistical purposes.

The '[]' symbol separates the two parallel statements and the '||' symbol separates the components of each statement. The SLOOP statement syntax is described in detail in Section 4.3.6.1[10]. The two parallel statements are executed infinitely often and in any arbitrary order. An operational view of a SLOOP statement execution is that all its components execute simultaneously. The semantics of a SLOOP statement is discussed in Section 4.3.6.3. For the purposes of explaining the example below it suffices to say that statement components extend the notion of multiple assignment statements. It enables one to specify multiple actions that can viewed as being executed simultaneously. **All the conditions specified in the various components of a statement are evaluated before any assignments are made.** Section 4.3.6.3 contains a discussion of the concept of simultaneous execution of message expressions.

```
   self transform: newElement
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength]
|| bufferedElements addLast: newElement
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength]
|| newElement := nil
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength]
|| maximumRecordedLength := bufferedElements size + 1
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength and:
      [bufferedElements size + 1 > maximumRecordedLength]]
                                                  "Statement S1"

[] consumer pass: (bufferedElements first)
      if consumer readyToReceiveElement and:
      [bufferedElements size > 0]
```

[10] Section 4.3.6.1 describes additional syntactical constructs that would facilitate a more succinct version of the statements shown here. The shorter version is presented in Section 4.3.6.3.

```
|| bufferedElements removeFirst
      if consumer readyToReceiveElement and:
      [bufferedElements size > 0]                    "Statement S2"
```

Statement *S1* describes the actions when a new element is available and the buffer has not yet reached its maximum length. The first component of *S1* performs the transformation on the new element, the second component appends the element to the buffer, the third component sets `newElement` to nil and the fourth component increments the `maximumRecordedLength` if the new length of the buffer is greater than any previously recorded length. Note that the conditional expressions of all the components of *S1* are evaluated before any assignments are made. Thus, for example, the value of `newElement` is only set to nil once its current value has been used in the evaluation of **all** the conditional expressions of *S1*.

Statement *S2* describes the actions when the consumer is ready to receive the next element and the buffer is not empty. The first component of *S2* passes the first element of the buffer to the consumer and the second component removes the element from the buffer.

The above program excerpt contains several examples of SLOOP message expressions. The message expression `bufferedElements size` is an example of a *unary message expression*. It comprises a *receiver* (`bufferedElements`) and a *selector* (`size`) that has no *arguments*. When `bufferedElements` receives this message, it returns the number of elements present in the buffer. The *message expression* `bufferedElements addLast: newElement` is an example of a *keyword message expression*. The latter is composed of one or more *keywords*, followed by an *argument* for each of the keywords. In this example there is only one keyword (`addLast:`). The argument is `newElement`. When `bufferedElements` receives this message, it appends `newElement` to the buffer.

The role of a message **expression** (such as `bufferedElements size`) in the above statements is similar to that of a terminating function in UNITY statements. Each UNITY statement is executed atomically, therefore all terminating functions associated with a particular UNITY statement form part of that atomic unit.

Similarly, each statement in the above example is executed **atomically**. The message expressions that are associated with a particular statement form part of that atomic unit. The reason for devising the concept of a **sequential** method is to be able to encapsulate a **sequence of statements that may not be interleaved** with others. This allows one to reason about the correctness of the statements within a sequential method without having to take any interleaving with statements in any other methods into account.

As was stated in Chapter 1, it is possible to represent any computation in terms of parallel statements only. However, the resulting program is likely to be at a lower level of abstraction, since the functionality of each sequential method would then be represented by one **or more** parallel statements. If a sequential method is represented via multiple parallel statements, it means that the sequential method is no longer executed atomically. The finer granularity of atomicity could make the program more complex. The purpose of the sequential method construct is therefore to enable the software designer to work at a higher level of abstraction.

For example, a method called `removeFirstTwoElements` of a `Queue` class is a good candidate to be designed as a sequential method. If written as such a method, the preconditions for executing the method would include the predicate that the object should contain at least two elements. The

`removeFirstTwoElements` method would then typically contain two sequential statements, each invoking the `removeFirst` method, as shown below:

```
  self removeFirst
[] self removeFirst
```

By virtue of the definition of a sequential SLOOP method there can be no interference by any other statements of the program while the `removeFirstTwoElements` method is executing.

If there had been no concept of a sequential method, the functionality of the `removeFirstTwoElements` method would have had to be achieved by implementing a mechanism in the `Queue` class to enable the sender to obtain exclusive rights to remove elements from the object.

One possible design to achieve this is shown below (the instance of the `Queue` class is called `buffer`).

First of all, the `buffer` object would have to maintain a variable called `objectRequestingRemoveElement`. The value of this variable would be nil if no object needs to remove an element from `buffer`. If an object needs to remove one or more elements from `buffer`, it needs to invoke the `objectRequestingRemoveElement:` method, which will then set the value of the `objectRequestingRemoveElement` variable to represent the identifier of the object that invoked the method. The precondition for setting the `objectRequestingRemoveElement:` variable to the new value is that its current value should be nil. Thus, a `buffer` object would only allow one object at a time to set this variable.

The `buffer` object will not allow the `objectRequestingRemoveElement` variable to be set to nil via this method. Instead, the `objectReleasingRemoveElement:` method should be invoked. The precondition of this method is that the current value of the `objectRequestRemoveElement` variable should match the value passed as the argument. The `buffer` object will only allow the object that has an identifier matching the value of the `objectRequestRemoveElement` variable to remove an element from the `buffer` (for example, the method to remove the first element of the `buffer` would require the identifier of the sending object to be passed as an argument).

Thus, any object needing to remove an element from `buffer` would first have to invoke the `objectRequestingRemoveElement:` method of `buffer`. Once the sender object has invoked the necessary methods to remove the relevant elements, the sender object has the responsibility to ensure that it invokes the `objectReleasingRemoveElement:` method to set the `objectRequestRemoveElement` variable to nil.

An object that needs to remove two **successive** elements from `buffer` would therefore have to contain the following parallel statements:

```
   buffer objectRequestingRemoveElement: self
      if buffer objectRequestingRemoveElement isNil and:
            [buffer size >= 2]
|| elementCounter := 1
      if buffer objectRequestingRemoveElement isNil and:
            [buffer size >= 2]                                  "CS_S1"
[] buffer removeFirst: self
      if elementCounter > 0 and: [elementCounter < 3]
|| elementCounter := elementCounter + 1
      if elementCounter > 0 and: [elementCounter < 3]           "CS_S2"
```

```
[] elementCounter := 0
      if elementCounter = 3
|| buffer objectReleasingRemoveElement: self
      if elementCounter = 3                                    "CS_S3"
```

From the above example it is clear that the inclusion of the sequential method construct enables the software designer to write parallel statements at a **higher level of abstraction**. Statements *CS_S1*, *CS_S2* and *CS_S3* can be replaced by the following statement if the concept of a sequential method is allowed:

```
buffer removeFirstTwoElements
      if buffer size >= 2
```

There would be no need for an `objectRequestingRemoveElement` variable and the `removeFirstTwoElements` method would simply be implemented via two sequential statements, each invoking the `removeFirst` method, as shown earlier in this section.

Thus, by allowing sequential methods in a SLOOP program, it **enables the software designer to group all the functionality that needs to be executed as an atomic unit into a single parallel statement**. Typically, the **complexity** associated with mutual exclusion is **reduced** by executing the statements associated with the critical section within a single parallel statement, as demonstrated by the `removeFirstTwoElements` example.

As mentioned in Chapter 1, another important benefit of allowing sequential methods in a SLOOP program is the fact that a large part of the extensive Smalltalk **class library** can be **reused** (only the classes and methods related to multiprocessing are excluded).

The **rationale** for devising **parallel methods** is presented next. Since SLOOP is an object-oriented method, the **structuring** and **encapsulation** features are also applied to those SLOOP statements that are allowed to interleave with one another. This is the reason for having **parallel** methods. These methods therefore contain those SLOOP statements that may be interleaved arbitrarily.

The terms "sequential statements" and "parallel statements" are required to differentiate between SLOOP statements that appear in sequential and parallel methods respectively. The two statements in the Producer-Consumer example given earlier in this section could therefore form part of a parallel method of a class that handles the processing of `bufferedElements`.

The reason for executing only one parallel statement whenever a parallel method is invoked, is to be able to execute the parallel statements of a program in an arbitrary order, as required by the computational model. For example, suppose program X contains class A and class B, and parallel method p_A belongs to class A and parallel method p_B belongs to class B. Method p_A contains two parallel statements, viz. p_A1 and p_A2. Method p_B also contains two parallel statements, viz. p_B1 and p_B2. If all the statements of a parallel method are executed each time the method is invoked, then the following execution order would not be possible:
p_A1, p_B1, p_A2, p_B2.

Although the computational model is based on a set of parallel statements that are executed in an arbitrary order, it will not affect the correctness of the program if certain execution orders are excluded, provided each parallel statement is executed infinitely often. However, by specifying that only one parallel statement of a parallel method should be executed at each invocation of that method, the fact that **the parallel statement is the unit of atomicity** in a SLOOP program is re-inforced.

As stated earlier, the arbitrary ordering of the execution of the parallel statements does not affect the correctness of the parallel method. **The correctness properties of the method are based on the fact that each parallel statement will be executed infinitely often and does not rely on any order of execution.** The conditions associated with the statements (if such conditions are required), ensure that the values of the variables referenced by the statements will be changed in the correct order.

Another reason for only executing one parallel statement at each invocation of a parallel method is to reduce the number of objects that need to be reserved before executing a parallel method. This issue will be explained further in Chapter 8.

The structure of a SLOOP program is described next in order to place the approach followed during the analysis and design phases in context.

4.3 The structure of a SLOOP program

The SLOOP syntax is given below in BNF. All terminal symbols are shown in plain or boldface type and the non-terminal symbols in italics. Braces are used for grouping. Square brackets indicate that the enclosed syntactical unit is optional. If a syntactical unit is followed by an asterisk, it denotes zero or more occurrences; a plus indicates one or more occurrences. Options are separated by vertical bars. Literals are enclosed by single quotes.

All names, such as *program-name*, *class-name*, *instance-name* and *category-name*, are strings that may contain letters, digits and the underscore character. They all start with a letter. If the plural form is used (for example as in *instance-variable-names*), then one or more name(s) may be present, each separated by a white-space character. Three consecutive colons separate a *class-name* from a *package-name* when it is necessary to qualify the *class-name* by a *package-name*. The triple colon symbol also separates consecutive *package-name*s in the case of nested packages.

4.3.1 The *SLOOP-program*

The **static** structure of a SLOOP program consists of an ***activation-section*** and **a collection of classes,** partitioned into one or more **packages**. SLOOP packages have an organisational function only. A package denotes a group of logically related classes. A class does not allow other classes belonging to the same package to access or modify any of its attributes in any way other than via the methods defined for that class. Thus, the data belonging to a class is fully encapsulated by that class. Each class may contain sequential and/or parallel methods.

Viewed **dynamically**, a SLOOP program first instantiates a number of classes and thereafter it executes a set of **parallel statements** infinitely often. These parallel statements are contained in methods belonging to the classes in the program and can only be executed infinitely often if their containing methods are invoked infinitely often. This requires the *a priori* instantiation of the classes to which these methods belong. The purpose of the *activation-section* is to ensure that these **classes are instantiated** and that the relevant **parallel methods are invoked infinitely often**.

Each *SLOOP-program* has the following structure (comments may be inserted anywhere in the program and are enclosed by double quotes):

SLOOP-program → **program** *program-name*
activation-section
{*package-description*}+
end-program

activation-section → **sequential**
statement-list
end-sequential

parallel
statement-list
end-parallel

package-description → **package** *package-name*
{*package-description*}*
{*class-description*}*
{*partial-class-description*}*
end-package

The BNF definition of a SLOOP *statement-list* is given in Section 4.3.6.1. Section 4.3.2 contains BNF definitions of a *class-description* and a *partial-class-description*.

Thus, a SLOOP program contains a number of statements to activate the system, followed by a description of all the constituent classes of the system.

The *activation-section* contains **sequential** and **parallel** statements following the corresponding keywords. The distinction between sequential and parallel statements located in the *activation-section* is similar to the distinction between such statements located in sequential and parallel methods. The sequential statements in the *activation-section* are executed only once and in the order of their appearance. Parallel statements, on the other hand, are executed infinitely often and in any order.

Since there are no containing methods for the statements located in the *activation-section*, these statements are **not** executed as a result of method invocations. Instead, these are the statements that execute when the program starts running. The sequential statements are executed first, followed by the infinitely often execution of the parallel statements.

The sequential statements in the *activation-section* are used to create the instances of the classes where such instances that need to exist before the parallel statements can start executing. Each parallel statement is executed infinitely often **by virtue of the fact** that it appears in the ***activation-section*** after the keyword **parallel**. These are the statements that invoke the parallel methods defined for the classes of the program. Parallel methods may be class or instance methods, but since a class method is usually used to create an instance of the class that it belongs to, it is unlikely that there will be a requirement to execute class methods infinitely often. Parallel methods are therefore usually instance methods.

Note that the invariants associated with a class form part of the preconditions of a parallel instance method. These invariants need only hold after the class has been instantiated and initialised. It is for this reason that the execution of the parallel statements in the *activation-section* may only commence once the sequential statements in this section have completed execution, i.e. sequential statements are used to establish class invariants.

When a system has been activated, it means that all the class instances that need to be operational at start-up time have been created and the relevant **parallel** methods of these class instances have been activated (i.e. they are being invoked infinitely often). Note that **all** classes **need not** be instantiated in the *activation-section*. Those classes that do not contain parallel instance methods[11] and that are not presumed to exist when the statements in the *activation-section* are executed, need not be instantiated at this stage. A class may contain multiple sequential and/or parallel methods. Not all of these methods need to be used within a specific application. For example, a UserAgent class may have a separate parallel method for each type of user supported by the system. Depending on the type of users applicable to a specific instance of the system, the appropriate parallel methods are activated.

At least one parallel instance method has to be invoked in the *activation-section*. This implies that at least one class has to be instantiated (the one to which the parallel method belongs). The sequential statements in the *activation-section* may also perform other functions, but they must at least perform this instantiation function.

Classes are grouped into one or more packages. As in the Unified Modeling Language (UML), the purpose of a package is to provide a convenient mechanism to refer to a group of model elements [RSC-Web]. Packages may be nested, i.e. a package contains class(es) and/or other package(s). Since the scope of a class name is restricted to the package in which it is contained, the package mechanism also facilitates support of multiple classes with the same name, provided those classes are located in different packages. The class name has to be prefixed with the appropriate package name if it is used in a package other than the one in which the class is defined. Since packages may be nested, package names can be chained together. SLOOP packages therefore provide a convenient **structuring** mechanism.

Each class referenced in the *activation-section* has to be present in one of the *package-descriptions*. A *package-description* may consist of other *package-description*(s), *class-description*(s) and/or *partial-class-description*(s).

The call centre example below illustrates the concepts introduced thus far. (Figure 4-3 provides a graphical representation of the problem domain.) This is the example which is used throughout the remainder of this thesis. It was chosen because it is non-trivial and can therefore be used to demonstrate the **scalability** of the SLOOP method. Due to the **diversity** of its classes many aspects of the SLOOP method can be illustrated via this example. Later chapters (i.e. Chapter 5 and onwards) show how the problem presented in this example can be "solved", i.e. taken through from the requirements analysis phase to the generation of executable code. Here various SLOOP concepts are illustrated based on aspects of a solution to the problem.

The following is a brief description of the problem (it is described in more detail in Chapter 5): Software for a call centre needs to be designed. At a high level of abstraction a call centre is a system which accepts telephone calls from service users, enters the associated service requests into the relevant queues depending on the service requested and finally assigns the service requests to the appropriate service providers.

For example: User X dials the toll-free number for service Y. The Public Switched Telephone Network (PSTN) switches the call to the Private Branch Exchange (PBX) at the call centre premises. The Interactive Voice Response prompts the caller to press specific numbers identifying the service

[11] The TimeoutElement class described in Appendix B, Section B.12, is an example of a class that does not have any parallel methods.

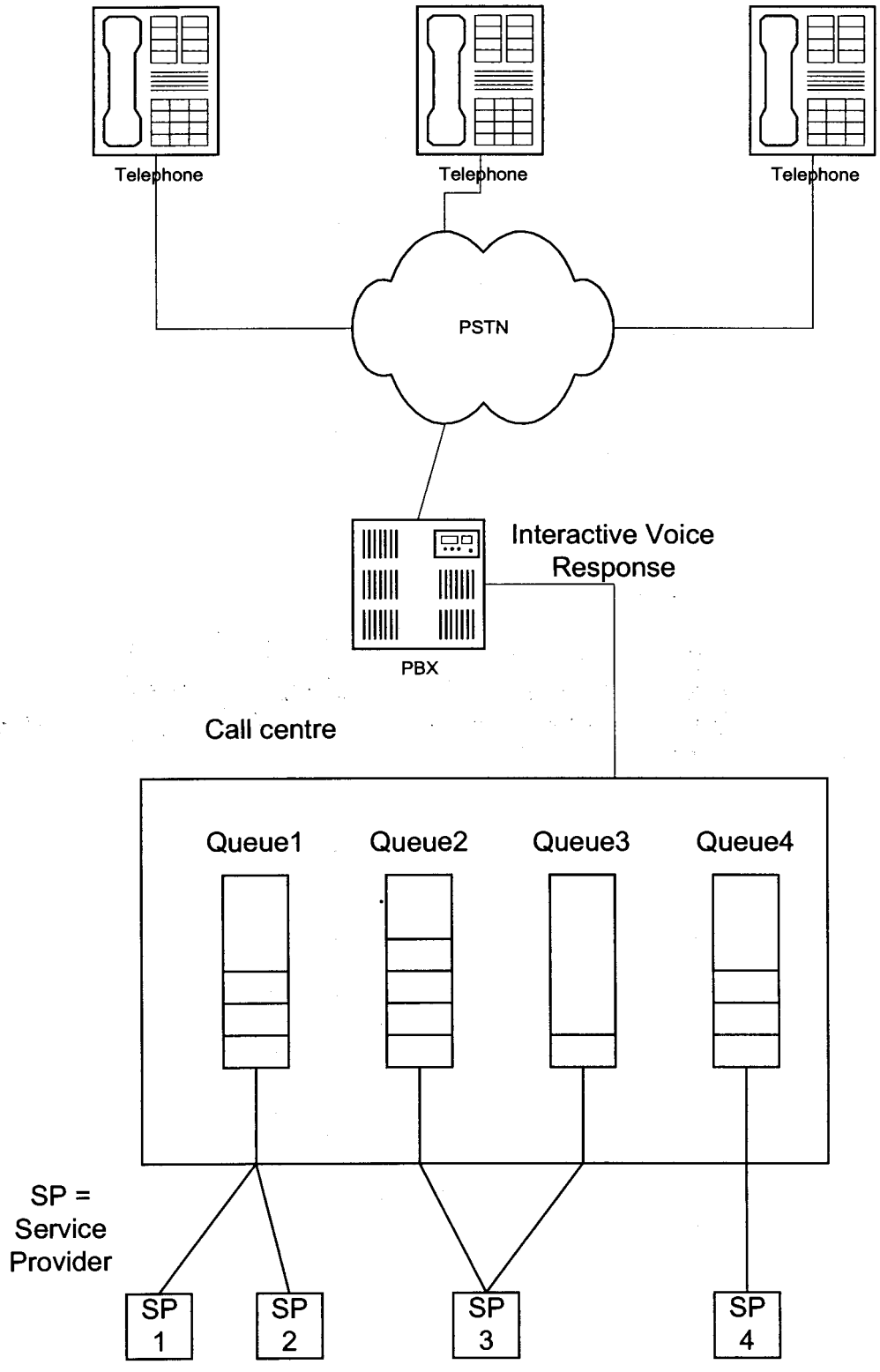

Figure 4-3. Call centre example.

desired. The relevant information pertaining to a particular call is referred to as the service request. The latter is passed to the system containing the call centre software.

The service request is added to the end of the appropriate service queue in the system, based on the service request category. When a service provider servicing that particular queue becomes available, the service request is removed from the queue and passed to the service provider. Note that some service queues are serviced by multiple service providers (e.g. in Figure 4-3 Queue 1 is serviced by both SP 1 and SP 2). A single service provider may also service multiple service queues (e.g. SP 3 services both Queue 2 and Queue 3 in Figure 4-3).

In this example a **simulation** program is developed in order to facilitate testing of the software without requiring participation from actual service users and service providers. Another reason for implementing a simulation program is that it allows one to produce a system which can be executed even when the implementation dependent details have not been specified yet. Thus, the product is executable without requiring any further subclassing; the implementation dependent classes are all defined as concrete classes using either simulations or defaults to handle the implementation dependent aspects. This approach also facilitates **rapid prototyping**. Thus, the simulation classes in the example below are **not** required in order to make the progression from design to implementation seamless; they are merely provided for the reasons given above.

The following program fragment exemplifies the high level structure of a SLOOP program. (Note that at this point the SLOOP method to arrive at this structure is not at issue.)

```
program CallCentreSimulation
        sequential
        aCCSimulationActivation := CC_SimulationActivation setup
        end-sequential
        parallel
        aCCSimulationActivation p_activate
        end-parallel


package CC_ActivationPkg
class CC_Activation
        "Remainder of SLOOP description of CC_Activation class"
        ...

class CC_SimulationActivation
        "Remainder of SLOOP description of CC_SimulationActivation class"
        ...

...     "SLOOP descriptions of other classes in the CC_ActivationPkg"
end-package


package CC_CorePkg
class ServiceRequest
        "Remainder of SLOOP description of ServiceRequest class"
        ...

...     "SLOOP descriptions of other classes in the CC_CorePkg"
end-package
```

```
package SystemUtilitiesPkg
class TimerServices
      "Remainder of SLOOP description of TimerServices class"
      ...

...   "SLOOP descriptions of other classes in the SystemUtilitiesPkg"
end-package

package CC_SimulationInterfacesPkg
class EventSimulator
      "Remainder of SLOOP description of EventSimulator class"
      ...

...   "SLOOP descriptions of other classes in the
      CC_SimulationInterfacesPkg"
end-package

package SmalltalkLibPkg
class OrderedCollection from SmalltalkLibRepository
      "Remainder of SLOOP description of OrderedCollection class"
      ...

...   "SLOOP descriptions of other classes in the SmalltalkLibPkg"
end-package

end-program
```

As shown in Figure 4-4, the `CallCentreSimulation` program contains five packages, viz. the `CC_ActivationPkg`, the `CC_CorePkg`, the `SystemUtilitiesPkg`, the `CC_SimulationInterfacesPkg` and the `SmalltalkLibPkg`. The `CC_ActivationPkg` contains all the activation classes of the program. The `CC_SimulationActivation` class has a method that activates all the classes required for a call centre **simulation.** There may be zero or more other activation classes containing methods activating the classes required for various types of **actual** call centres. All classes with such methods inherit from the `CC_Activation` class, which handles the activation of the classes that are **independent** of the type of call centre being constructed. The label "incomplete" in Figure 4-4 indicates that other subclasses of the `CC_Activation` class have not yet been specified, but should still be defined.

The `CC_CorePkg` contains all the classes that remain unchanged regardless of whether a simulation or an actual system is being constructed. The `SystemUtilitiesPkg` contains classes that are generally useful to many applications. For example, a `TimerServices` class allows a client of the class to set a timer and then informs the client when the timer has expired. The `CC_SimulationInterfacesPkg` contains the simulation classes. In an actual system this package would be replaced with the `CC_InterfacesPkg` containing the classes representing the actual interfaces to the service users and service providers. The `SmalltalkLibPkg` contains the classes that are found in existing Smalltalk libraries.

The classes that need to be instantiated for the system may be instantiated directly by the sequential statements in the *activation-section* of the program. Alternatively, one or more classes in the packages of the program may perform that function. In this example the `CC_ActivationPkg` contains the `CC_SimulationActivation` class that instantiates all the classes required by the simulation system. This illustrates that in the SLOOP context the package modelling element has no function other than to provide a collective name to a group of related classes. It is not necessary for

each package to contain its own activation class. The classes of a package may be instantiated from within another package. In this example only the `CC_SimulationActivation` class therefore needs to be instantiated directly[12] by the sequential statements of the *activation-section*. All the other classes are instantiated indirectly via the instance of the `CC_SimulationActivation` class.

The parallel methods that need to be activated for the system may be invoked directly from the parallel statements in the *activation-section* of the program. In this example, the activation is achieved by sending the message `p_activate` to the instance of the `CC_SimulationActivation` class.

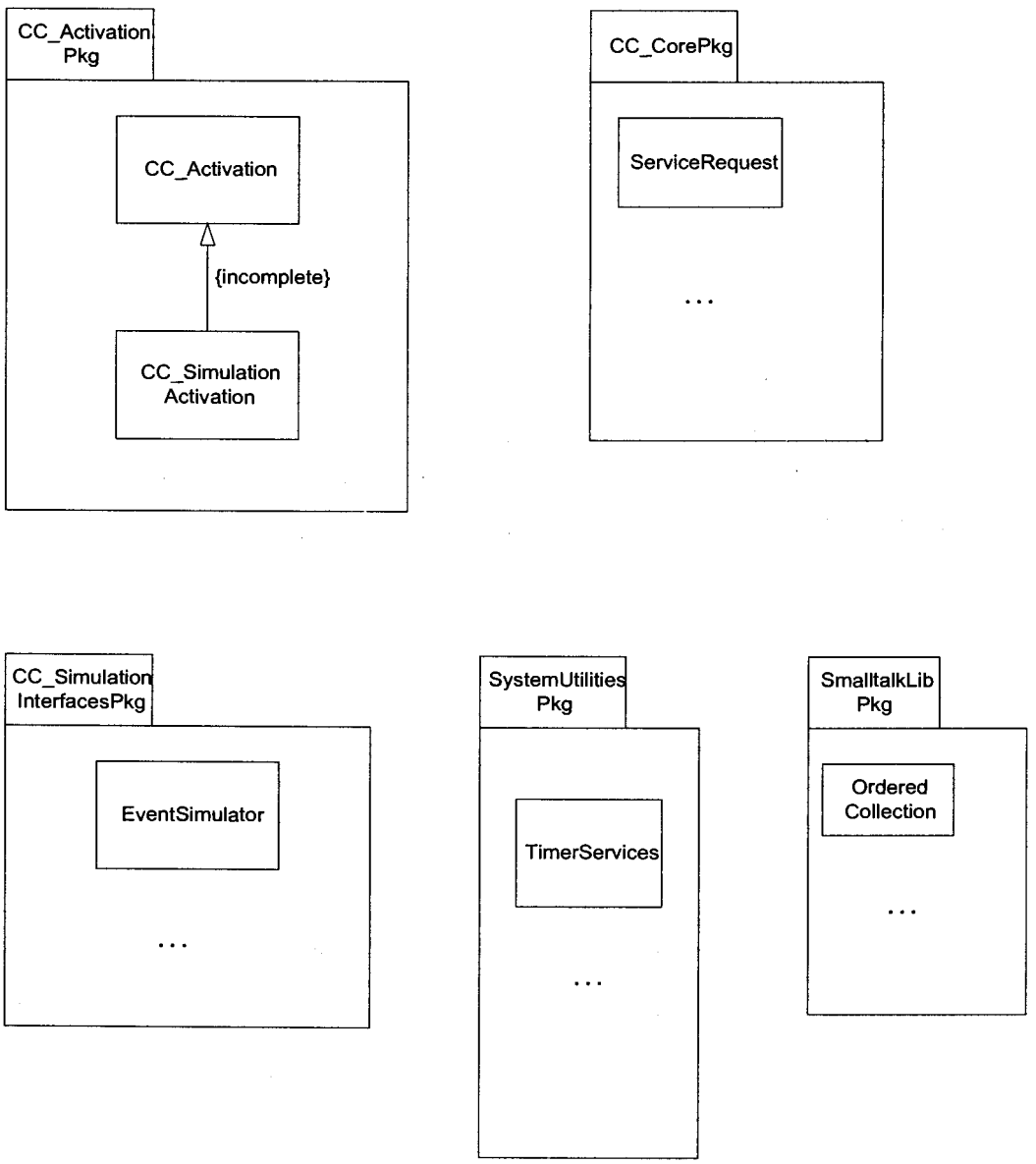

Figure 4-4. Package structure of the call centre.

[12] This is achieved by sending the 'setup' message to the CC_SimulationActivation class.

Partial-class-descriptions are used to list those classes that are obtained from a repository, i.e. classes that are being **reused**. A **partial-class-description** only contains the name of the class, the name of the repository where it can be found and a list of the methods that are used by the current system. A *partial-class-description* does not repeat any other information (e.g. correctness properties) from the full *class-description* in the repository. This is to ensure the **integrity** of the specification, i.e. the full *class-description* is the only place where it appears and where it is updated. Note that a *partial-class-description* is **not** an **interface description** of a class (e.g. like the short form that is used to specify class interfaces in Eiffel [Meye97]). Instead, the purpose of a *partial-class-description* is to specify which methods of a class that is specified elsewhere are being reused by this particular program.

A class that is not obtained from a repository needs to be specified in full, in which case the *class-description* is used. A class may be stored in a repository in a stand-alone SLOOP *class-description*. A detailed discussion of a *partial-class-description* and a *class-description* is presented next.

4.3.2 SLOOP classes

A SLOOP class is described by its name, its superclass, a list of all its class and instance variables, the class macros and correctness properties, as well as its class and instance methods. The valid combinations of these fields can be deduced from the following BNF description:

class-description →
class *class-name*
superclass *superclass-name* [**from** *repository-name*]
[**class variable names** [*class-variable-names*]]
[**instance variable names** [*instance-variable-names*]]
[**class macros** [*macros-section*]]
class properties [*properties-section*]
methods-section

A *partial-class-description* has the following syntax:

partial-class-description →
class *class-name* **from** *repository-name*
partial-class-methods-section

The **class properties** keywords are mandatory in order to emphasise to the designer that all the relevant properties should always be listed. If this section had been optional, then it would have been unclear to the person reusing the class whether the original designer had merely chosen not to list the properties or whether there had been no relevant properties to list.

The *macros-section*, *properties-section* and *methods-section* are now described in more detail.

4.3.3 The *macros-section*

4.3.3.1 Purpose of the macros-section

The *macros-section* comprises one or more *macro-definitions*. Each *macro-definition* defines a *macro-variable* in terms of a *macro-expression*. A *macro-definition* therefore allows for one

variable (the *macro-variable*) to be written in terms of one or more others. Thus, the *macro-name* can be used as a shorthand form for the *macro-expression*. It is a **notational convenience.**

Note that the *macro-variable* is **not an attribute** of the class. It may also not appear on the left-hand side of any sequential or parallel assignment statement. Each time the *macro-variable* is referenced, the *macro-expression* is evaluated. Since a *macro-variable* may be used in each component of a SLOOP statement (and all components execute simultaneously), **the *macro-expression* should have no side-effects**, otherwise the result of the execution of the statement would be non-deterministic. Another reason why it may have no side-effects is because the predicates in the *properties-section* of a class or method may also refer to *macro-variables*. Since the correctness properties are at a meta level, i.e. they describe rather than modify the behaviour of the system, the evaluation of *macro-variables* within these correctness properties should not cause any modifications to the system state.

A *macro-expression* may contain another *macro-variable*, but only if the latter is defined earlier in the list of *macro-definitions*. This ensures that there are no circular definitions. A *macro-variable* may only appear once on the left-hand side of a *macro-definition* in a *macros-section*.

4.3.3.2 The syntax

The syntax of the *macros-section* is as follows:

macros-section	→	*macro-list*
macro-list	→	*macro-definition* {[] *macro-definition*}*
macro-definition	→	*macro-variable* ≡ *macro-expression*
macro-variable	→	*variable-name*
macro-expression	→	*simple-macro-expression* | *conditional-macro-expression*
simple-macro-expression	→	*message-expression* | *variable-name* | *literal*
conditional-macro-expression	→	*simple-macro-expression* if *boolean-expression* {~ *simple-macro-expression* if *boolean-expression*}*

A *message-expression* is a Smalltalk-style message expression and a *boolean-expression* is a Smalltalk-style message expression that returns true or false. A Smalltalk-style message expression consists of a *receiver*, a *selector* and zero or more *arguments*. If there are no arguments, the message is called a *unary message*. For example, `bufferedElements size` is a unary message expression. A *binary message* has a single argument following a selector consisting of one or two non-alphanumeric characters, the second of which may not be a minus sign. The message expression `a + b` is an example of a binary message expression. The third type of message is a *keyword message*. The selector consists of one or more keywords, each with its associated argument. A keyword consists of an identifier followed by a colon. For example, `bufferedElements addLast: newElement` is a keyword message expression.

A *literal* is a Smalltalk-style literal which may be a number, a symbol constant, a character constant, a string or an array constant.

Note that, as in Smalltalk-80, message expressions may be nested. The receiver of a message expression may itself be a message expression. Similarly, the argument(s) of a keyword message may also be message expression(s).

A message expression may also contain a block. The reader is referred to [GoRo89] for a detailed discussion of a Smalltalk-80 block. For the purposes of explaining its usage in the examples in this thesis, the following description suffices.

A block represents a deferred sequence of actions. It consists of a sequence of expressions separated by periods and delimited by square brackets. The actions represented by a block are not necessarily executed when the block expression is encountered.

For example, a block expression is used as the argument of the Smalltalk-80 `and:` keyword message. This message represents the logical "and" operation. If the first operand (i.e. the receiver) evaluates to true, the second operand (the argument of `and:`) is evaluated and its value is returned as result. However, if the first operand evaluates to false, the second operand is not evaluated. This is indicated syntactically via the fact that the argument of the `and:` message is a block expression.

A block is also used when the receiver of a message is a collection and the actions represented by the block need to be applied to each element of the receiver. In that case each element of the receiver is passed to the block as an argument. This is indicated syntactically by the presence of an identifier preceded by a colon at the beginning of the block. This identifier is separated from the rest of the contents of the block by a vertical bar.

For example, the Smalltalk-80 library `select:` and `detect:` messages are used frequently in the CallCentreSimulation example. The `select:` message evaluates the block received as argument of the message for each of the receiver's elements (the receiver is a collection object). It returns a collection that contains only those elements of the receiver for which the block evaluates to true. The message expression below returns the collection representing all employees earning a salary greater than $20 000:

```
employees select: [:each | each salary > 20000]
```

The following message expression returns the object representing the first employee found earning a salary greater than $20 000 (if no such employee is found, then nil is returned):

```
employees detect: [:each | each salary > 20000] ifNone: [nil]
```

For a formal description of the Smalltalk-80 syntax, the reader is referred to [GoRo89].

In the BNF definition of the *macros-section* as shown above the "[]" symbol separates multiple *macro-definitions* and the "~" (tilde) symbol separates the various cases of a *conditional-macro-expression*. These symbols were chosen because they are used in a similar context in UNITY. By following this approach the learning curve is reduced for those already familiar with UNITY.

One of the conditions of a *conditional-macro-expression* **must** evaluate to true, otherwise the result of the evaluation of the *macro-expression* will be undefined. If more than one condition evaluates to true, the *simple-macro-expressions* corresponding to the cases that evaluated to true must all result in the same value, so that the execution of a *macro-definition* is always deterministic[13]. For example, the following macro-definition is not allowed:

[13] This is similar to the requirement in UNITY that every assignment should be deterministic [ChMi88]. (Dijkstra's guarded commands [Dijk76] do not have this requirement).

```
x ≡ 1  if y > 0 ~
   -1  if y > 0 ~
    0  if y ≤ 0
```

The rationale for this requirement is as follows: In SLOOP programs non-determinism is modelled via the computational model, i.e. all parallel statements execute infinitely often and in any order. The statements may be conditional. Thus, at any stage any statement can be executed and if its conditions evaluate to true, then the corresponding actions are performed. For example, the following SLOOP statements demonstrate this non-determinism:

```
  x := 1     if y > 0 ~
  x := 0     if y ≤ 0
[] x := -1   if y > 0 ~
  x := 0     if y ≤ 0
```

When reasoning about the correctness of a SLOOP program, the software designer has to take this non-determinism into account. If non-determinism is allowed in *macro-definitions* as well, the number of **levels** of non-determinism is increased. This adversely affects the **understandability** and **maintainability** of the software, especially since a *macro-expression* is not visible in the parallel statement itself. There is no need for this increased complexity, since the computational model is powerful enough to model any kind of non-determinism.

As will be shown in Section 4.3.5, a method may have its own *macros-section* associated with it. A *macro-definition* appears in the **class** *macros-section* if it is used by more than one class or instance method and if it does **not** refer to *pseudo-variables*[14] used in one or more message patterns of the class. It appears in the *macros-section* of a **method** if it is used by that method only or if it references a *pseudo-variable* of that method.

The scope of a *macro-definition* appearing in a **method** *macros-section* is local to the method. The same *macro-variable* may therefore be defined in more than one method and the associated *macro-expressions* may evaluate to different values in the different methods. However, if a *macro-variable* is defined in the **class** *macros-section*, it may not be defined in any **method** *macros-section*. Again, this restriction is made for the sake of **understandability**.

4.3.3.3 Example usage

The following example demonstrates the usage of the *macro-definition*. It also shows how *macro-definitions* are inherited by subclasses. In the `CallCentreSimulation` program described in Section 4.3.1 an activation class called `CC_SimulationActivation` was introduced. The function of this class is to instantiate all the classes required by the system upon startup. Its parent class, `CC_Activation`, contains the methods to instantiate the classes that are common to all call centres. This functionality forms part of the `initialize` method[15] of the `CC_Activation` class. The `CC_SimulationActivation` class contains some additional methods to instantiate the classes that are specific to a call centre simulation. The following *macro-definitions* form part of the class *macros-section* of the `CC_Activation` class (the `CC_SimulationActivation` class inherits both the *macro-definitions* and the `initialize` method):

[14] A message pattern consists of a selector and the set of pseudo-variables that represent the arguments of the message. Thus, the pseudo-variables are used within the method to refer to the arguments of the message.

[15] The CC_Activation and CC_SimulationActivation classes are fully specified in Appendix B, Sections B.2 and B.3 respectively.

```
class macros
        maxConn ≡ config maximumConnections
                "Number of simultaneous user connections supported"
        ...     "Other class macros"
```

Each telephone call mentioned in the example in Section 4.3.1 is modelled by an instance of the `Connection` class. Each call centre has a configurable maximum connection capacity. This maximum is configured via `config`, an instance of the `Configuration` class, which also forms part of the call centre system. This value is obtained from `config` by sending the message `maximumConnections` to it. Whenever an iteration over all the connections in the system has to be performed, this value is used. It is convenient to refer to the maximum number of connections as `maxConn` in the quantifications. The alternative is `config maximumConnections`.

For example, in the *properties-section* of the `initialize` method of the `CC_Activation` class, it is stated that all instances of the `Connection` class will exist after the method has completed. The object containing all the `Connection` instances is called `userConnections` in this example. Each `Connection` instance may be referenced by using an index into the `userConnections` collection. The predicate contains the following expression:

```
... ∧
< ∀ i where 1 ≤ i ≤ maxConn :: (userConnections at: i) notNil
> ∧ ...
```

The alternative expression would have been:

```
... ∧
< ∀ i where 1 ≤ i ≤ config maximumConnections :: (userConnections at:
i) notNil
> ∧ ...
```

The macro defining the `maxConn` variable is a class macro. It can therefore be used in the properties and statements of all the methods of the `CC_Activation` class and its subclasses. As shown in the `initialize` method below, the macro is not defined within this method as well, since the scope of the macro is class-wide (the `userConnections` and `config` variables are instance variables[16] of the `CC_Activation` class and are therefore also known within this method):

```
message pattern initialize
        method properties
        "Total correctness"
        true results-in[17]
                ... ∧
                userConnections notNil ∧
                < ∀ i where 1≤i≤maxConn :: (userConnections at: i )
                notNil
                > ∧ ...
```

[16] Instance variables are created per class instance. This is as opposed to class variables, which are shared amongst all the instances of a class. Instance variables exist for the entire lifetime of a class instance and are therefore persistent from one method invocation to the next for a specific class instance.

[17] The **results-in** logical relation will be explained in Section 4.3.4.4.

```
sequential
...
[] userConnections := SmalltalkLibPkg:::Array new: maxConn
[] < [] i where 1≤i≤maxConn :: userConnections at: i
      put: (self initConnection: i)
  >
...
end-sequential
```

Figure 4-4 also contained a `TimerServices` class. The `p_runTimer:` method of the `TimerServices` class contains a macro that illustrates the usage of a *conditional-macro-expression*:

```
difference ≡ currentTime - lastTime
               if (currentTime - lastTime) ≥ 0 ~
             currentTime + (86400 - lastTime)
               if (currentTime - lastTime) < 0
```

The `TimerServices` class continually checks whether a timer tick has elapsed since the time was last recorded in `lastTime`. The most recent time is recorded in `currentTime`. The `lastTime` and `currentTime` variables contain the time converted into seconds since midnight. Provision therefore has to be made for the case where the `currentTime` could have a smaller value than `lastTime`. This example clearly demonstrates the convenience of being able to refer to the *macro-variable* rather than the *macro-expression* in those statements that need to calculate the difference between the `currentTime` and the `lastTime`.

4.3.4 The *properties-section*

4.3.4.1 The syntax

Each class, as well as each method within a class may contain a *properties-section*. The *properties-section* lists the correctness properties that describe the behaviour of the class or method. It contains safety, liveness and/or precedence properties. The syntax of a *properties-section* is as follows:

properties-section → *property-list*
properties-list → *property*
{[] *property*}*

A *property* may be of the form:
p **unless** *q*,
stable *p*,
invariant *p*,
p **ensures** *q*,
p **leads-to** *q*,
p **until** *q*,
p **detects** *q*,
p **precedes** *q* or
p **results-in** *q*,
where *p* and *q* are **first-order** predicates.

Note that message expressions are allowed as part of the predicates used in the SLOOP logical relations. Such a message expression may not have any side-effects, i.e. it may not cause any permanent change to the state of the system. The reason for this restriction is obvious: Correctness properties describe a system at a meta level. These properties are **about** the system and should certainly not modify the system that they are describing.

The reason for allowing message expressions in the SLOOP logical relations is to make the incorporation of correctness properties into a SLOOP program as **understandable** and **seamless** as possible. The predicates used in the correctness arguments resemble the message expressions used in the SLOOP statements themselves.

In addition, it enables one to specify properties at a higher level of **abstraction**. For example, instead of specifying that `newElement = nil` should hold before the `newElement` attribute can be set to another value, the precondition of the `newElement:` method could contain the expression `self canAcceptNewElement`. Thus, instead of referring explicitly to the values of the attributes of the object, a method is invoked which determines whether the `newElement` attribute can be set to another value.

Such an approach would allow subclasses to redefine the conditions for accepting a new element without having to modify the property specification of the `newElement:` method. The capability to do this is essential in order to support **polymorphism**. The client continues to invoke the `canAcceptNewElement` method regardless of whether it is dealing with the parent class or one of its descendants. This is similar to the approach followed in the Eiffel object-oriented language, where the assertions may contain function calls [Meye97]. (In the Eiffel language a function is a method which returns a value and which may have no side-effects.)

By allowing message expressions in the predicates, one can therefore take advantage of the benefits of **encapsulation** and **information hiding** even at the level of property specifications. As illustrated by the example above, when the properties of a class refer to the attributes of another class, then those attributes are not accessed directly, but rather via the methods provided by the target class.

A further advantage of allowing message expressions in the predicates is the fact that it facilitates **reuse** of correctness properties. This aspect is described in more detail in Section 4.3.4.2.

The above logical relations are of a temporal nature. Their definitions will be given in Section 4.3.4.4. The present section only deals with the syntax.

Since **first order predicate logic** is used, universal and existential quantification is allowed in the logical relations. The keywords **forall** and **exists** may be used as alternatives to the $\forall$ (universal quantification) and $\exists$ (existential quantification) symbols respectively. Instead of using a colon to denote the domain of a quantification, the reserved word **where** is used. This is to avoid confusion with the colon used in Smalltalk keyword expressions.

If a *variable-list* is used in a quantification, the variables are separated by a comma preceded by a backslash. This ensures that the comma cannot be mistaken for the Smalltalk concatenation symbol. If a '<' symbol is followed immediately by a quantification symbol, it denotes the start of a quantification construct. If a '>' symbol appears as the first non-white-space character on a line, it denotes the end of a quantification construct. The quantification constructs have the same parsing precedence as parentheses.

The $\wedge$ (logical and), $\vee$ (logical or) and $\neg$ (negation) operators are defined in addition to the Smalltalk & (logical and), | (logical or) and not (negation) operators. The additional logical operators serve a readability purpose only. One difference between the additional logical operators and the Smalltalk ones is in the parsing precedence. The Smalltalk unary, binary and keyword expressions are evaluated in that order. The additional logical operators are evaluated after the unary, binary and keyword expressions have been evaluated. This results in fewer parentheses as illustrated by the example below. The example is one of the liveness properties of

the `start:id:for:` method of the `TimerServices` class (one of the classes in the `SystemUtilitiesPkg` discussed in Section 4.3.1).

Whenever a new timer is started, the `TimerServices` instance creates a new instance of the `TimeoutElement` class to represent this particular timer. The `TimeoutElement` instance (`nextElement` in the example) is then entered into a collection of timers represented by `timeoutCollection at: writeIndex`. (The `TimerServices` class design is described in detail in Appendix B, Section B.11.) The property below states that if a timer with a duration greater than zero is requested, then `nextElement` will be inserted into the relevant entry of `timeoutCollection` and the postconditions of the `TimeoutElement` instance creation method will hold.

Without additional logical operators:

```
"Total correctness"
0 < duration & (duration ≤ maximumTimeout) results-in
      methodReturnValue = self &
      (nextElement class = TimeoutElement) &
      ((timeoutCollection at: writeIndex) includes:nextElement)&
      (TimeoutElement postconditions: (#setup:id:for:)
      withArguments: #(requestor identifier duration))
```

With additional logical operators:

```
"Total correctness"
0 < duration ∧ duration ≤ maximumTimeout results-in
      methodReturnValue = self ∧
      nextElement class = TimeoutElement ∧
      (timeoutCollection at: writeIndex) includes: nextElement ∧
      TimeoutElement postconditions: (#setup:id:for:)
      withArguments: #(requestor identifier duration)
```

Another difference between the additional logical operators and the corresponding Smalltalk operators is that the additional operators are non-evaluating[18], i.e. if the first operand evaluates to false in the case of conjunction, the second operand is not evaluated. Similarly, if the first operand evaluates to true in the case of disjunction, the second operand is not evaluated. Non-evaluating conjunction is especially important when a boolean expression contains a test for the existence of an object in conjunction with an invocation of one of its methods. If the test for the existence of the object fails, no method will be invoked on it. The SLOOP ∧ and ∨ operators should be mapped to the Smalltalk non-evaluating `and:` and `or:` selectors during an implementation in order to ensure that no message is sent to a non-existing object in this type of situation.

The following are further conventions about the priorities of logical relations (those on the same line have equal priority and the lines represent the priorities from high to low):

¬

=, ≠

∧, ∨

⇒

≡

unless, ensures, leads-to, stable, invariant, detects, until, precedes, results-in.

Properties are universally quantified over all the free variables occurring in them.

[18] This is Smalltalk-80 terminology [GoRo89]. In C, C++ and Java the corresponding term is short-circuit evaluation.

4.3.4.2 Reuse of correctness properties

When a statement within a method sends a message to another object and the correctness properties of the target method have significance in the correctness properties of the sending method, then the SLOOP notation provides constructs to highlight such reuse. In such a case the properties of the sending method contain either of the following expressions, depending on whether the target selector requires arguments or not:
target-object postconditions: (*#target-selector*)
or
target-object postconditions: (*#target-selector*) withArguments: *target-arguments*.

If arguments are required, the arguments are provided in the form of an array in *target-arguments*, as illustrated in the example in the previous subsection. The correctness property in that example represents a liveness property of the `TimerServices` class. The property is of the form *p* **results-in** *q*. One of the conjuncts in *q* is the following:

```
TimeoutElement postconditions: (#setup:id:for:)
            withArguments: #(requestor identifier duration).
```

This conjunct represents the predicate that will hold when the `setup:id:for:` method of the `TimeoutElement` class has successfully[19] completed execution and if `requestor`, `identifier` and `duration` are used as arguments for the respective keywords of the selector. Thus, in addition to the postconditions that will hold as a result of assignments to attributes of the `TimerServices` class instance, the postconditions of the `setup:id:for:` method of the `TimeoutElement` class will also hold when the `start:id:for:` method of the TimerServices class has completed execution.

This approach towards correctness property reuse is followed to avoid the specification of the same correctness properties in multiple places, with the associated risk of **inconsistency**. Furthermore, this notation allows one to indicate syntactically that correctness properties are being **reused**. In the SLOOP method correctness properties are not proved from first principles each time they are being used. The correctness arguments for each class and its methods are given only once. Thereafter the method is assumed to behave correctly, provided its preconditions are satisfied when it is invoked.

4.3.4.3 Class properties and method properties

Correctness properties can be specified for a class and also for each individual method. The nature of the method (i.e. whether it is sequential or parallel[20]) dictates the way in which the correctness properties are interpreted. This is due to the fact that a sequential **method** is executed as an atomic unit (usually as part of the execution of a parallel statement), whereas the individual **statements** of a parallel method are executed atomically. Furthermore, the conventional model of control flow applies to the statements of a sequential method, while parallel statements are executed infinitely often and in any order. The interpretation of class and method correctness properties given next is based on these differences.

The **safety** properties specified in the **class** *properties-section* must be preserved by each instance method of the class. This means that these properties are required to hold immediately before the execution of the first statement of any sequential instance method and immediately after the program has returned from the execution of a sequential instance method. The safety properties specified in the **class** *properties-section* also have to hold before and after the execution of each

[19] This requires that the preconditions of the setup:id:for: method should hold when the latter commences execution.
[20] Sequential and parallel methods were defined in Section 4.2.3.

statement of each parallel instance method. These properties only become effective after instance creation has taken place, i.e. they have to hold after the execution of the last statement of an instance creation method, but they do not have to hold before that.

The **liveness** properties specified in the **class** *properties-section* must be satisfied by the parallel instance methods defined for that class, i.e. the specified progress must be ensured by the infinitely often execution of the statements within the parallel methods defined for that class. These properties do not apply to the sequential methods defined for that class.

The properties specified for **individual methods** are considered next.

Since a **sequential method** can be viewed as a terminating sequential program, the correctness of a sequential method is sufficiently described via a **total correctness** property specified for that method. A total correctness property of a sequential method specifies that the method should **terminate** and that it should produce the **correct results** when it terminates.

Thus, the total correctness property of a sequential method specifies that, provided the corresponding preconditions are met when the first statement of the method is executed, control will return to the client and when it returns, the postconditions specified for the property will be satisfied. It is the responsibility of the client to ensure that the preconditions of the method are satisfied before the method is invoked. The behaviour of the method is undefined if the preconditions are not met. Note that the **class** invariants are **implicit** pre- and postconditions, since they have to hold before and after each sequential method is executed.

The correctness properties of a **parallel method** are **not** based on the execution of a **single** invocation of that method. Although the execution of each invocation of a parallel method terminates, a parallel method must be invoked infinitely often in order to ensure that each parallel statement is executed infinitely often as dictated by the computational model. The correctness properties are therefore based on the assumption that each parallel statement within the parallel method will be executed infinitely often and in any order. For such a computational model it is more appropriate to define safety and liveness properties that are quantified over all the statements of the parallel method.

Thus, the **safety** properties of a **parallel method** must be preserved by each statement of the parallel method. A safety property is therefore **universally quantified** over all the statements of the method. A **liveness** property of a **parallel method** specifies that if the precondition of that property is satisfied at some stage, then the postcondition will subsequently be satisfied, provided all the statements of the method are executed infinitely often. If the precondition is never satisfied, the execution of the method will have no effect. The liveness properties of a parallel method are **existentially quantified** over all the statements of the parallel method.

4.3.4.4 Definitions of the logical relations

In Chapter 2, Section 2.5.5, the UNITY logical relations were defined in terms of the execution of any UNITY multiple assignment **statement** *s* in a **program** *F*. In the case of a SLOOP program, correctness properties are specified for **classes and their methods**. The definitions of the SLOOP logical relations reflect the fact that different computational models are used for sequential and parallel methods.

The logical relations pertaining to parallel methods are defined in terms of the *statements* of parallel methods. In contrast, all the statements of a sequential method are always executed as an atomic unit, therefore the *unless* logical relation in the **class** *properties-section* does not refer to the individual sequential statements, but rather to the sequential method as a unit. The correctness properties of sequential methods are described in terms of total correctness properties.

In the definitions below the phrase "$\forall s$ *where* s in *PM*" refers to any statement s that forms part of the set of statements that make up the parallel method *PM*.

In the class *properties-section*:

For a given class *C*, where *PM* is any parallel method of class *C* and *SM* is any sequential method of class *C*,

p **unless** $q \equiv <\forall s$ **where** s in $PM :: \{p \wedge \neg q\}\ s\ \{p \vee q\}$
$> \wedge$
$<\forall SM :: \{p \wedge \neg q\}\ SM\ \{p \vee q\}$
$>$

For a given class *C* and all parallel methods *PM* of class *C*,

p **ensures** $q \equiv <\exists\ PM$ **where** *PM* is a parallel method of class $C ::$
$(p$ **unless** $q \wedge <\exists\ s$ **where** s in $PM :: \{p \wedge \neg q\}\ s\ \{q\}$
$>)$
$>$

In the sequential method *properties-section*:

For a given class *C* and a particular sequential method *SM* of class *C*,

p_{entry} **results-in** $q_{exit} \equiv (p_{entry} \Rightarrow \Diamond$ at $\text{location}_{\text{exit}} \wedge q_{exit})$

where p_{entry} represents the preconditions for the sequential method, $\Diamond$ is the *eventually* temporal logic operator, $\text{location}_{\text{exit}}$ represents an exit location and q_{exit} represents the postconditions.

In the parallel method *properties-section*:

For a given parallel method *PM* of class *C*,

p **unless** $q \equiv <\forall s$ **where** s in $PM :: \{p \wedge \neg q\}\ s\ \{p \vee q\}$
$>$

For a given parallel method *PM* of class *C*,

p **ensures** $q \equiv (p$ **unless** $q \wedge$
$<\exists\ s$ **where** s in $PM :: \{p \wedge \neg q\}\ s\ \{q\}$
$>)$

The **leads-to** logical relation can be derived from the inference rules as specified in Chapter 2, Section 2.5.5. The other logical relations are defined in terms of the three basic logical relations, viz. **unless**, **ensures** and **leads-to**. The derivation rules are the same for both UNITY and SLOOP and were given in Section 2.5.5 of Chapter 2.

4.3.4.5 Method properties and the value returned by a method

As in Smalltalk-80, each method always returns a value. If there is an explicit return value, it is indicated by the '^' symbol, otherwise the method returns the receiver. In order to refer to the value returned by the method, the special variable `methodReturnValue` is used in the method *properties-section*. The scope of the variable is restricted to the *properties-section* of the specified method. It is not used in the statements of the method. The total correctness property of a method which returns a value always refers to `methodReturnValue` in its postconditions.

4.3.5 The *methods-section*

Each class description in a SLOOP program contains a mandatory *methods-section*.

4.3.5.1 The methods-section syntax

The syntax of the *methods-section* of a class is as follows:

methods-section → **[class methods**
{*methods-implementation*}+]
[instance methods
{*methods-implementation*}+]

methods-implementation → **category** *category-name*
{*method-description*}+

method-description → *sequential-method* | *parallel-method*

sequential-method → **message pattern** *Smalltalk-message-pattern*
[method macros *macros-section*]
method properties *properties-section*
sequential
statement-list
end-sequential

parallel-method → **message pattern** p_*Smalltalk-message-pattern*
[method macros *macros-section*]
method properties *properties-section*
parallel
statement-list
end-parallel

partial-class-methods-section
→ **[class methods**
{*selector*}+]
[instance methods
{*selector*}+]

selector → *Smalltalk-selector* |
p_*Smalltalk-selector*

The distinction between class and instance methods is the same as in Smalltalk, i.e. class methods are those that are handled by the class, whereas an instance of the class responds to an instance method [GoRo89]. For example, the Smalltalk `OrderedCollection` class responds to the `new` class method by creating an instance of itself. This instance can then respond to instance methods such as `addLast:`, which will append an element to the end of the collection.

The methods are categorised into different **categories**. The purpose of these categories is similar to the purpose of the message categories in Smalltalk-80, viz. the categories are intended to make the description of the methods more readable. For example, the `testing` category is used for methods that return true or false depending on the values of certain variables. The `cyclic` category is used for parallel methods. The name "cyclic" reflects the nature of the method, viz. that its statements execute infinitely often. The SLOOP method does not prescribe what category names should be used.

A *Smalltalk-message-pattern* has the usual Smalltalk syntax, i.e. it comprises the message selector with the associated pseudo-variables to represent the arguments if there are any. A

Smalltalk-selector has the usual syntax of a selector in a Smalltalk program. The SLOOP *statement-list* is discussed in Section 4.3.6.

Similarly to the properties of the class, all the relevant properties of the method should be listed if there are any. If this section had been optional, then it would have been unclear to the person reusing the class whether the original designer had merely chosen not to list the properties or whether there had been no relevant properties to list.

Parallel methods are distinguished from sequential methods by the *p_* prefix that is attached to parallel selectors.

The *partial-class-methods-section* is used in a *partial-class-description*. When a class from a repository is being reused, only those methods that are being reused in the new program should be named in the latter. The selectors that identify these methods are present in the *partial-class-methods-section*.

4.3.5.2 Method invocation

All methods are **invoked** synchronously, i.e. the client only continues execution once the invoked method has returned. However, in the case of a sequential method, the postconditions of the method can be assumed to hold when the method returns, whereas the liveness properties of a parallel method need not necessarily be verifiable when it returns control to the calling method. Parallel method **execution** therefore has **asynchronous** semantics.

The liveness properties of a parallel method assume that the method is invoked infinitely often and that all the statements within the method are thus executed infinitely often. However, **eventually** (after some finite number of invocations) the properties should indeed be verifiable. For example, for some parallel method *PM* the property *p* **ensures** *q* might not be verifiable after the first invocation of *PM*, but eventually it will be. The concepts of synchronous and asynchronous invocations as well as synchronous and asynchronous semantics were discussed at length in Chapter 3, Section 3.2.2.1.

4.3.5.3 Parallel method activation and the number of active parallel statements

The difference between the parallel statements in the *activation-section* and those in a parallel method is that only one statement in a parallel method is selected at each invocation. The method is invoked infinitely often. In contrast, an *activation-section* is executed only once and the parallel statements in the *activation-section* are executed infinitely often during the execution of the *activation-section*. Thus, once an activation has completed its sequential statements it goes into an infinite loop where it executes its parallel statements. It is irrelevant whether only one or all the parallel statements are executed per loop iteration, as long as each parallel statement is executed infinitely often.

The number of multiple assignment statements in UNITY may not vary dynamically, because it is difficult to reason about statements that are part of the program infinitely often instead of all the time [ChMi88]. For example, it might happen that whenever a statement is present, then it is not scheduled. In general, SLOOP programs have the same restriction, but with the following exception to the rule:

If system A contains a parallel statement which has an *if* clause, then the execution of such a parallel statement only has an effect if the condition specified in the *if* clause evaluates to true. If system B is designed in such a way that the statement is not present in the list of parallel statements whenever this condition evaluates to false, but it is present whenever this condition evaluates to true, then the behaviour of the two systems is equivalent.

The following hypothetical example illustrates the concept. Suppose parallel method `p_incrementCounter` of class `X` in system A contains the following parallel statement:

```
counter := counter + 1   if input > 0
```

Clearly the instance variable `counter` is only incremented if the `input` instance variable is greater than 0. The statement in the `Activation` class instance which invokes the `p_incrementCounter` method is as follows (`anX` is an instance of class `X`):

```
anX p_incrementCounter
```

System B is similar to system A, but in this case the parallel statement in the `p_incrementCounter` method is not conditional:

```
counter := counter + 1
```

Instead, the statement in the `Activation` class instance which invokes the `p_incrementCounter` method is conditional (when the `input` message is sent to `anX`, the value of the `input` instance variable is returned):

```
anX p_incrementCounter   if anX input > 0
```

Thus, if the value of the `input` instance variable is less than or equal to zero, then the `counter := counter + 1` statement does not form part of the set of parallel statements executed by the program in system B, since its containing method is not invoked.

It is clear that the results produced by systems A and B are the same. Effectively, the condition which controls the incrementing of `counter` has merely been moved. Instead of being part of the parallel statement itself, it now forms part of the condition which determines whether the method containing the parallel statement should be invoked.

A more detailed discussion of the usage of **dynamic parallel statements** is given in Chapter 9 when the application of the State design pattern in the SLOOP context is discussed. In that chapter it will be shown that this feature is necessary in order to cater for certain types of designs.

4.3.5.4 Nesting of SLOOP method invocations

The differences between sequential and parallel methods and statements and how they may be combined were described in Section 4.2.3. In short, sequential methods contain sequential statements and they may invoke sequential methods only. Parallel methods contain parallel statements and they may invoke both sequential and parallel methods. Figure 4-5 illustrates the various combinations graphically. In order to keep the diagram uncluttered, the receivers of the messages are not shown.

First of all the sequential statement in the *activation-section* is executed. It invokes the `setup` method of the class that instantiates all the other classes in the system. The `setup` method is a sequential method containing 2 sequential statements. The two statements invoke sequential methods `method1` and `method2` respectively.

Once the sequential statement in the *activation-section* has completed execution, the parallel statements are executed in an infinite loop. In this example there is only one statement, viz. the one containing the invocation of the `p_activate` method. The `p_activate` method contains two parallel statements containing invocations of `p_method1` and `p_method2` respectively.

In turn, `p_method1` contains 3 parallel statements. These statements contain invocations of `p_method3`, `p_method4` and `p_method5` respectively. Finally, `p_method3` contains a single

parallel statement that invokes `method3`, a sequential method. A parallel method that contains parallel statements invoking sequential methods only, is called a **leaf parallel method**.

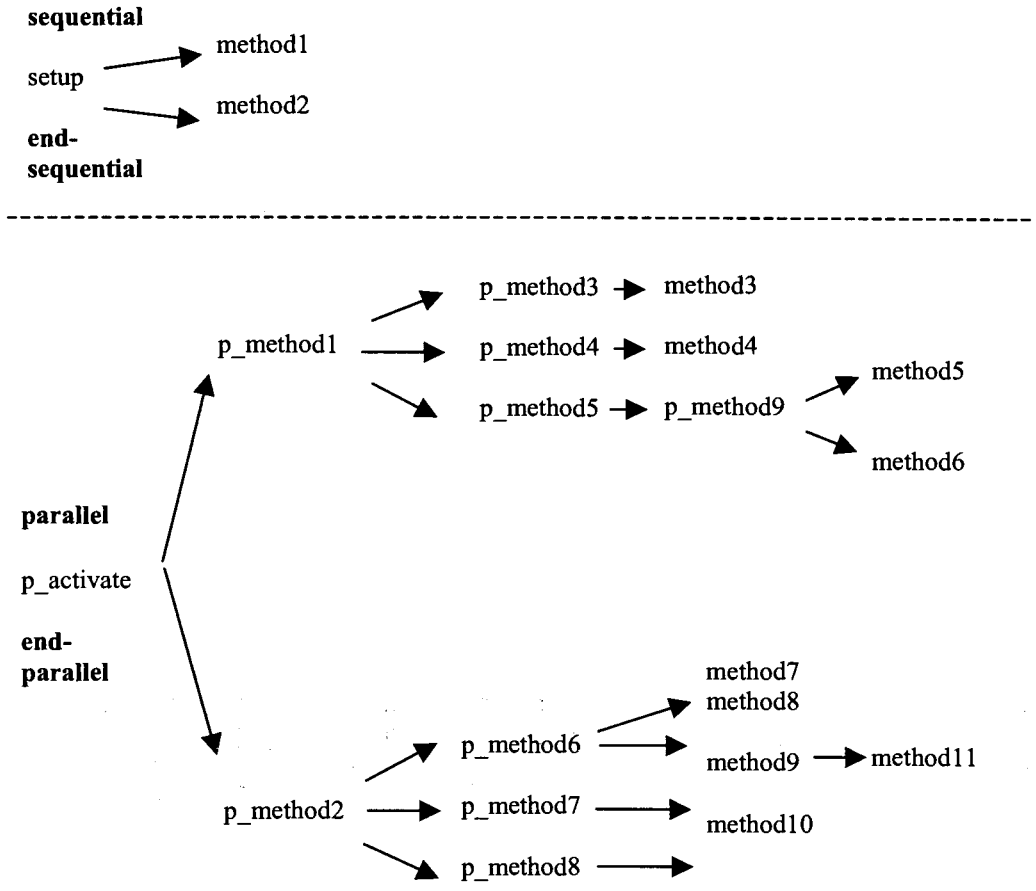

Figure 4-5. Example combinations of methods from different categories.

When comparing the SLOOP categorisation of statements with the UNITY approach, the parallel statements have the same purpose as the assignment statements in a UNITY program. A UNITY assignment statement may invoke any terminating function. Similarly, a SLOOP parallel statement may invoke any sequential method, since the latter has to terminate and the postconditions have to be satisfied when it returns.

However, in UNITY there is no concept of a statement that calls functions containing other parallel statements. Program structuring is via the union of the statements of two programs and via superposition, i.e. via transforming and adding statements in a specific way as described in Chapter 2. Thus, there is no nesting of UNITY statements; they all execute at the same level.

A SLOOP program could be viewed as a collection of parallel statements that invoke terminating sequential methods only. This differs from the UNITY approach in the way that these leaf parallel statements may be referred to, i.e. these statements may be grouped and a collective name given to them (the encapsulating parallel method). In turn, the encapsulating method may be contained in a parallel statement of another method, thereby creating a hierarchical structure. Allowing a parallel statement to invoke another parallel method in a SLOOP program provides a mechanism to avoid having to list every leaf parallel statement of the program in the *activation-section*. It enables one to make use of the **structuring** features that are inherent in an object-oriented approach.

The leaf parallel statements are the only ones that may invoke **modifying** sequential methods, i.e. they are the only ones that result in permanent changes to the state of the system[21]. The purpose of this restriction is to aid **understandability** and **reliability**. Modifications to the system state need not be checked for at every level of the hierarchy of statements; they should only appear in leaf parallel statements. The statements at higher levels of the hierarchy should therefore not have unexpected side-effects. This restriction can be compared with the disallowing of the goto statement; it is not essential, but it makes the program easier to understand and reason about.

The various statements in a non-leaf parallel method do not all have to result in the same number of nesting levels before their respective leaf methods are reached. This is illustrated in Figure 4-5: The parallel method `p_method1` contains 3 parallel statements. These statements invoke `p_method3`, `p_method4` and `p_method5` respectively (i.e. `p_method1` invokes parallel methods only). Sequential methods `method3` and `method4` are invoked by `p_method3` and `p_method4` respectively. However, `p_method5` invokes yet another parallel method `p_method9`, which then invokes sequential methods. Thus, each of the first two statements in p_method1 invokes a leaf parallel method, whereas the third statement only reaches a leaf parallel method via another parallel method.

A sequential method that is invoked by a leaf parallel method may invoke other sequential methods, as illustrated by sequential `method9` in Figure 4-5.

The notion of a program that comprises a set of parallel statements, where these statements execute independently of each other, fits in well with the object-oriented paradigm. A number of parallel methods may be defined for a class and the relevant methods can be activated if required by the application.

The parallel statements that belong together in a SLOOP program may be grouped together for structuring purposes, but any statement can be added to the list of parallel statements executed by a program by creating a new method and activating it. Although it is possible to override one of the original parallel methods of the class in order to add the new statement, it is not a requirement if the original statements are not modified or removed. This demonstrates an aspect of the **flexibility** of the SLOOP method.

4.3.5.5 SLOOP objects and events

Once the relevant parallel methods of a SLOOP object have been activated, they are invoked infinitely often while the program is running. Although a parallel statement executes infinitely often, it may have **conditions** associated with it, implying that the statement only has an effect on the state of an object if the conditions are met. Such a condition could be the occurrence of an **event.** This means that the objects react to events via their parallel statements.

In Chapter 3, Section 3.2.1, the issues surrounding the representation of multiple threads of control in object-oriented programs were discussed. It is clear that the SLOOP approach provides an elegant solution to the problem. Any object that executes parallel statements, can react to events. **Concurrency is therefore implicit in the design**. Whether or not the object is eventually mapped to a separate process or processor, is an implementation issue.

The **high level of abstraction** of the SLOOP method is also evident from the following: Since the parallel statements form part of parallel methods that are invoked infinitely often, the object is not executing these methods continuously. It can also execute a sequential method when it

[21] The parallel statements in the leaf parallel method may also contain assignments; i.e. modifications to the system state are not made only via sequential methods. This is evident from the SLOOP statement syntax, which will be given in Section 4.3.6.1.

receives the relevant message from a client. At the design level it is not necessary to concern oneself with the way in which messages to the object would be integrated with events. This is handled when the program is mapped to a target architecture during the implementation phase.

An example of an object which responds to events as well as to messages from other objects is given in Appendix B, Section B.11. It describes the `TimerServices` class that is included in the `CallCentreSimulation` program. The parallel statements of this class check whether the timers that have already been started have expired, i.e. timer expiry is an event. However, a new timer is started by simply sending the message `start:id:for:` to the `TimerServices` instance. At the design level it suffices merely to specify the necessary parallel methods to handle the events and to provide the relevant sequential methods in order to respond to the messages that can be received. The mapping of SLOOP programs to various target architectures is described in Chapter 8.

4.3.6 The SLOOP *statement-list*

4.3.6.1 The syntax

The *statement-list* is defined as follows:

statement-list	→	*statement*[22]	
		{[] *statement* }*	
statement	→	*simple-statement* |	
		quantified-statement-list	
quantified-statement-list	→	<[] *quantification statement-list*>	
quantification	→	*variable-list* **where** *boolean-expr* ::	
variable-list	→	*variable* {\, *variable*}*	
simple-statement	→	*statement-component*	
		{ || *statement-component* }*	
statement-component	→	*enumerated-component*	
		| *quantified-component*	
enumerated-component	→	*component-part*	
		| *conditional-component-part-list*	
quantified-component	→	<|| *quantification simple-statement*>	
component-part	→	{[^]*variable* := *simple-expr*}[23] |	
		[^]*message-expression*	
conditional-component-part-list			
	→	*simple-component-part-list*	if *boolean-expr*
		{~ *simple-component-part-list*	if *boolean-expr* }*
simple-component-part-list	→	*component-part*	
		{\+*component-part*}*	
simple-expr	→	*message-expression* | *primary*	

A *message-expression* is a Smalltalk-style message expression and a *boolean-expr* is a Smalltalk-style message expression that returns true or false. A Smalltalk-style message expression was defined in Section 4.3.3.2. A *primary* is a Smalltalk-style primary which may be a variable name, a literal or a block. When a '<' symbol is immediately followed by the '[]' or '||' symbol, it denotes a quantification and the '<' symbol is not interpreted as a Smalltalk operator. If a '>' symbol appears as the first non-white-space character on a line, it denotes the end of a

[22] This implies that, as in UNITY, a statement-list cannot be empty.
[23] The braces around the [^]*variable* := *simple-expr* construct serve to identify the latter as a syntactic unit. This is needed because the '|' symbol has a higher precedence than the ':=' symbol.

quantification construct. A summary of the BNF definition of the SLOOP syntax can be found in Appendix A.

The restrictions regarding the use of Smalltalk library classes were given in Section 4.2.2. To recapitulate: no messages related to the Smalltalk-80 support for multiple processes may be used, since there is no concept of a process in a SLOOP program.

The SLOOP statement syntax is now discussed in terms of sequential and parallel methods.

4.3.6.2 Statements, components and parts

Exactly the same syntax is used for *statement*s in all the method categories. The '[]' symbol is merely a ***statement* separator**. It has no significance regarding the way in which the *statement*s are executed. That is determined by the keyword **sequential** or **parallel** preceding the *statement-list*. This also applies to the other symbols that are used in the SLOOP *statements*. The structure of a SLOOP statement is now described, followed by a discussion of the interpretation of the various symbols based on whether they are located in a sequential or parallel method.

*Statement*s consist of one or more ***statement-component*s**. The *statement-component*s are separated by the '||' symbol. Each *statement-component* consists of a ***component-part*** or one or more ***conditional-component-part-list*s**.

A *conditional-component-part-list* has an *if* clause associated with it. The *component-part*s in the list are only executed if the *if* clause evaluates to true. *Conditional-component-part-list*s are separated by the tilde '~' symbol, while the *component-part*s within the lists are separated by the '\+' symbol. The rationale for including these different constructs in the SLOOP syntax will be given later in this section. Examples will also be given to elucidate their usage. The SLOOP statement structure is illustrated in Figure 4-6.

The symbols found in **parallel** statements are interpreted in the following way: Each statement (demarcated by the statement separator symbol '[]') executes as an atomic unit. The statement execution may be interleaved in an arbitrary fair order with any other parallel statement execution in the system. The purpose of these statements is to **model parallelism** via the interleaving model [MaPn81a], which was described in Section 2.3.1 of Chapter 2. Since each statement executes independently of every other parallel statement, it also models **asynchrony** [ChMi88].

In contrast, the *statement-component*s of a parallel statement (separated by the '||' symbol) all execute simultaneously, thereby facilitating the **modelling of synchrony**. For example, during the mapping of a SLOOP program to a synchronous shared memory architecture, each *statement-component* of a particular parallel statement can be assigned to a separate processor. At each clock tick (there is a common clock in such an architecture), each processor executes a single computation step. If the SLOOP program is not mapped to a synchronous shared memory architecture, the execution is not necessarily simultaneous, but the same effect is achieved by performing the evaluation of the message expressions in a specific way. This is described in detail in the next subsection. The term "simultaneous execution" as used in the discussions below refers to both cases.

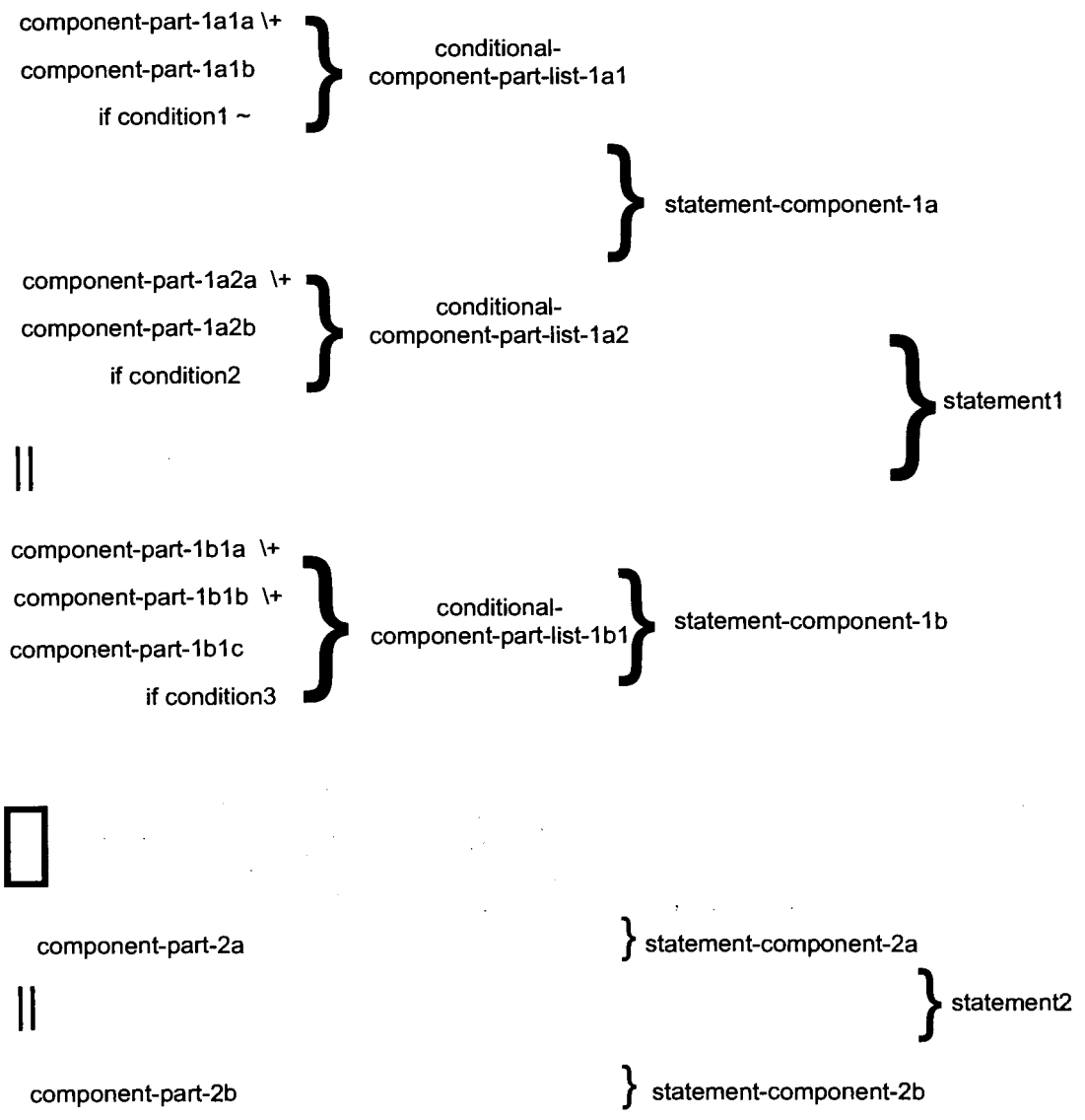

Figure 4-6. SLOOP statement structure.

The other purpose of having multiple *statement-component*s in a parallel statement is to be able to group all the computations that need to be executed as an atomic unit into a single statement. Since some of these may be conditional computations, it is convenient to decompose a *statement-component* even further into either a *component-part* or one or more *conditional-component-part-list*s. Each *conditional-component-part-list* contains all the computations that need to be carried out under the associated condition. All the *component-part*s within the *conditional-component-part-list* are executed **simultaneously**. The execution of a *component-part* depends on whether the *if* clause associated with the list in which it appears, evaluates to true.

The above concepts are now elucidated by an example. The following *statement* from a fictitious class consists of one *statement-component* containing a single *conditional-component-part-list*. The latter contains two *component-part*s, separated by '\+'. The conditional expression (represented by the *if* clause) controls the execution of both these *component-part*s.

```
a := a + 1 \+
b := exampleInstance number
      if exampleInstance notNil and: [exampleInstance number > 0]
```

Thus, if `exampleInstance` exists and if it returns a value greater than zero when the `number` message is sent to it, then `a` is incremented and `b` receives the return value of the `number` message.

If the *statement* is written as shown below, it has different semantics.

```
   a := a + 1
|| b := exampleInstance number
      if exampleInstance notNil and: [exampleInstance number > 0]
```

Here it comprises two *statement-component*s. The first *statement-component* contains a single *component-part* and the second one contains a *conditional-component-part-list*. The *conditional-component-part-list* contains a single *component-part*. Only the *component-part* in the second *statement-component* is subject to the evaluation of the conditional expression; the *component-part* in the first *statement-component* is always executed when the *statement* is selected for execution.

The *statement* below is an example of a *statement* containing multiple *conditional-component-part-list*s. The *statement* comprises two *statement-component*s. The first *statement-component* has two *conditional-component-part-list*s, separated by the '~' symbol. These lists represent the alternative values that `a` and `b` receive based on the value of `c`. The second *statement-component* contains a single *component-part* which is executed unconditionally. (Since all *if* clauses of a statement are evaluated before any of the *component-part*s are executed, the value of `c` in the *if* clauses will be the value before it is incremented. The evaluation order of SLOOP statements will be discussed in detail in the next section.)

```
   a := a + 1 \+
   b := b + 2
      if c > 0 ~
   a := a - 1 \+
   b := b - 2
      if c ≤ 0
|| c := c + 1
```

Note that the decomposition of a *statement-component* into *conditional-component-part-list*s and of the latter into *component-part*s is included in the notation in order to **emphasise relationships and dependencies**. The following statement, which does not make use of the '~' and '\+' constructs, is equivalent to the one above. In this case there are five *statement-component*s.

```
   a := a + 1
      if c > 0
|| b := b + 2
      if c > 0
|| a := a - 1
      if c ≤ 0
|| b := b - 2
      if c ≤ 0
|| c := c + 1
```

The earlier version of the above statement highlights the fact that there are two alternatives based on the value of `c` (there are two *conditional-component-part-list*s separated by a '~'). Furthermore, it emphasises the fact that the values of both `a` and `b` depend on the value of `c` (each *conditional-component-part-list* contains two *component-part*s separated by a '\+').

As discussed in Section 4.3.3.2, the value that is assigned to a *macro-variable* may depend on various conditions. The tilde notation is also used in a *conditional-macro-expression* to separate these alternatives.

The above format of *conditional-component-part-lists* (i.e. where the *component-part-list* is followed rather than preceded by the *if* clause) was chosen in order to keep the format similar to the UNITY notation. Since the SLOOP method is in the spirit of UNITY, it was decided to follow this approach throughout the development of the method.

All the *statements* in a **sequential** method are executed in sequence. The sequential method is executed as an atomic unit, i.e. the statements within the method are not interleaved with any other statements. Although multiple *statement-components* may be present, it does not have a specific purpose in a sequential statement. All the *statement-components* are executed **in their order of appearance**. The same applies to the *conditional-component-parts*, provided their associated *if* clauses evaluate to true. If an *if* clause evaluates to false, the associated *component-parts* are not executed and control is transferred to the next *conditional-component-part-list* if there is one. The purpose of having *component-part-lists* is to be able to associate a condition with multiple *component-parts*.

When an *if* clause is evaluated (in both sequential and parallel statements), the evaluation terminates when the first false condition is encountered if a non-evaluating **conjunction** is used as in the example below. Thus, if `exampleInstance` does not exist, no other expression in the statement is evaluated.

```
a := a + 1 \+
b := exampleInstance number
      if exampleInstance notNil and: [exampleInstance number > 0]
```

If a non-evaluating **disjunction** is used and the receiver evaluates to true, the *if* clause evaluates to true without any evaluation of the other operand. Should an *if* clause evaluate to false, none of the expressions in the corresponding *component-parts* are evaluated.

Any statement may return a value via the return operator, represented by the caret symbol ('^'). However, it is only meaningful for sequential methods. A parallel method has asynchronous semantics, which means that the client cannot expect its liveness properties to hold after a particular execution. They are only guaranteed to hold if the method is invoked infinitely often.

Note that a sequential method returns immediately after a *component-part* containing the return operator has been executed. For example, in the following *statement-list* the last four statements in the list are not executed if `exampleQ` contains no elements.

```
  ^ false
      if exampleQ isEmpty
[] workingElement := exampleQ first
[] workingElement message1
[] workingElement message2
[] ^ true
```

A *component-part* of a SLOOP statement may only contain a single (possibly nested) message expression. Cascaded message expressions (i.e. the receiver is not repeated in a sequence of message expressions if they all use the same receiver) are therefore not allowed in the SLOOP syntax. The following is a Smalltalk-80 example of a cascaded message expression:

```
Transcript cr; show: 'Testing'; cr.
```

The above cascaded message expression, which displays a carriage return followed by the word "Testing" and another carriage return, is equivalent to the following Smalltalk-80 message expressions:

```
Transcript cr.
Transcript show: 'Testing'.
Transcript cr.
```

Since these are three separate message expressions, a cascaded construct cannot be used in SLOOP statements. A SLOOP statement is restricted to contain a single message expression in order to **limit** the **complexity** of the statement.

4.3.6.3 Evaluation order

The evaluation rules for a UNITY multiple-assignment statement are fairly simple: all expressions on the right-hand side of the assignment and all subscripts on the left-hand side are evaluated first, followed by the simultaneous assignment of these values to the corresponding variables on the left-hand side of the assignment. Since the expressions on the right hand side of the assignment symbol may not modify any variables, they may be executed simultaneously or in any order. In turn, the assignment of the computed values may occur simultaneously or in any order.

In order to ensure that the synchrony as described in the previous section can be achieved, it is necessary to extend the above principle to the message expressions in SLOOP parallel statements. However, since the message expressions can be nested, the issue of the evaluation order of the expressions is rather more complex than in the case of UNITY multiple assignment statements.

The following examples illustrate SLOOP message expressions that are at various nesting levels.

```
inputQ addLast: newElement                                    "Example 1"
inputQ addLast: ((userConnections at: i) serviceRequest)      "Example 2"
(userConnections at: i) terminate: 'completed'                "Example 3"
```

The first example, which adds a new element to the end of the queue called `inputQ`, contains no nested expressions. The invocation of the method with the selector `addLast:` causes the assignments within this method and those within the methods called by `addLast:` (if there are any) to be executed.

The second example demonstrates that the argument(s) of a message expression may also be message expression(s). In this case the new element that is added to `inputQ` is obtained by sending the `serviceRequest` message to element `i` of the `userConnections` collection, i.e. the `ServiceRequest` instance associated with the `Connection` instance at index `i` of the `userConnections` collection is appended to `inputQ`.

The receiver of a message expression may also be a message expression as shown in the third example. It sends the `terminate:` message to the object at index `i` of the `userConnections` collection. (The `terminate:` method is described in Appendix B, Section B.7.)

The important issue in the case of UNITY is the fact that the evaluation of the expressions either on the right hand side or in subscripts on the left hand side of the assignment symbol **may not have any side-effects** and their **evaluation** must also be **completed** before any assignments are performed.

In order to ensure that a SLOOP statement (which may contain message expressions) has similar semantics, the following evaluation order applies to SLOOP parallel statements:
1) *If* clauses are evaluated first.
2) Thereafter all message expressions that form part of an argument of another message expression are evaluated. All message expressions that play the role of a receiver of a message are also evaluated at this time. Thus, the values of all receivers and arguments are obtained during this step. If the *component-part* contains an assignment symbol, all message expressions within that *component-part* are evaluated; the only action that is not performed is the actual

assignment to the instance or class variable. The order in which the receivers and arguments of a message expression within a *component-part* are evaluated is the same as for a Smalltalk-80 message expression.
3) Finally, the assignments in all *component-part*s are performed. If a *component-part* does not contain an assignment, the outermost message expression in that *component-part* is executed, i.e. the message expression resulting from the evaluations performed in step 2 is executed.

If the *if* clause associated with any *component-part* evaluates to false, no further computation is performed for that particular *component-part*.

In order to keep as close as possible to the model of the multiple assignment statement, the software designer has to observe the following rules:
Rule A: Methods that are executed as part of steps (1) and (2) should not modify any class or instance variables that are referenced by methods in other *component-part*s of the same statement.
Rule B: Methods that are executed as part of (3) should not modify any class or instance variables that are referenced by methods that are executed as part of (3) in other *component-part*s of the same statement.

The above evaluation steps are now demonstrated via a statement from the transformer example first presented in Section 4.2.3. (Recall that the purpose of the statement is to transform `newElement` and to add it to the `bufferedElements` queue if the latter has not yet reached its maximum length. It also keeps a record of the maximum length ever reached by the queue.)

```
   self transform: newElement \+
   bufferedElements addLast: newElement \+
   newElement := nil
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength]
|| maximumRecordedLength := bufferedElements size +1
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength and:
      [bufferedElements size + 1 > maximumRecordedLength] ]
                                                    "Statement S1"
```

All *if* clauses of all *conditional-component-part-list*s of a particular statement are evaluated before any of the *component-part*s are executed. All evaluations of `bufferedElements size` in statement *S1* will therefore yield the same result. Similarly, all evaluations of `newElement notNil` will yield the same result. Once the *if* clauses have been evaluated, all the message expressions as listed in step 2 are evaluated, followed by the evaluation of all message expressions and assignments as described in step 3.

As part of step 2 the following values are obtained (each line in the list below contains the list of values obtained for one of the *component-part*s in the above statement):
`self, newElement,`
`bufferedElements, newElement,`
`newElement, nil` and
`maximumRecordedLength, (bufferedElements size + 1).`

In the above list the order within each line is significant, but the order of the lines themselves is arbitrary.

As part of step 3 the following assignments and message expressions are executed (in any arbitrary order):

```
self transform: newElement
bufferedElements addLast: newElement
newElement := nil
maximumRecordedLength := bufferedElements size + 1
```

Since `newElement` was evaluated in step 2, the assignment of the value nil to the variable `newElement` in the third *component-part* does not affect the first or second *component-part*s. Similarly, because the value of `bufferedElements size + 1` was determined in step 2, this assignment is not affected by the execution of the `bufferedElements addLast: newElement` *component-part*. (The latter adds an element to the `bufferedElements` queue, thereby implicitly incrementing the size of the queue.)

In UNITY it is the **software designer's responsibility** to ensure that **a variable that appears on the left hand side of the assignment symbol in multiple components of the same statement is assigned the same value in all of these components [ChMi88]**. In SLOOP no variable may appear on the left hand side of the assignment symbol in multiple components of the same statement. This restriction is similar to that described in [Meye90]. Furthermore, Rule B described above must also be adhered to (methods that are executed as part of step 3 of the evaluation of a SLOOP parallel statement should not modify any class or instance variables that are referenced by methods that are executed as part of step 3 in other *component-part*s of the same statement).

4.3.6.4 The simplification of reasoning about correctness

The **atomicity** of SLOOP parallel statements is of paramount importance. In Chapter 2 it was stated that this atomicity, together with its considerable **expressive power**, facilitates software design at a higher level of abstraction. In turn, that simplifies correctness reasoning. The justification for that claim will now be given.

The previous subsections bear testimony to the expressive power of the SLOOP statement. The purpose of devising such a construct is to be able to achieve a higher level of abstraction. This is due to the following:

- Message expressions are allowed in the statements. One therefore designs in terms of classes and their methods. The execution of a method is at a higher level of abstraction than an assignment to a variable (as in UNITY).
- Since each parallel statement executes atomically, the designer can group all the actions that need to be performed without interference from other objects into a single parallel statement. There is therefore no need to design in terms of semaphores in order to control access to critical sections.
- When synchrony needs to be modelled, the fact that *statement-component*s within a single parallel statement are executed simultaneously provides the necessary expressive power.

By viewing the execution of each parallel statement as an atomic action, reasoning about programs is simplified, since only the pre- and postconditions of the sequential methods invoked by a parallel statement are relevant. Sequential method statements cannot interfere with any other statements in the program, since all the statements in the sequential methods invoked by a parallel statement form part of the atomic execution of the parallel statement. (The atomicity of a parallel statement also includes the evaluation of the *macro-expression*s referenced by *macro-variable*s in the statement. In addition, it includes the evaluation of its method and class properties.)

The **interleaving model** of concurrency [MaPn81a] was discussed in Chapter 2. In the SLOOP context this means that if two parallel statements share objects, then they are executed in some arbitrary order. If they do not share objects, they can be executed simultaneously. (Two parallel statements share objects if they send messages to the same objects, or if the two parallel

statements belong to the same object or if the one parallel statement sends a message to the object to which the other parallel statement belongs. Objects may be shared either explicitly via references in the parallel statement itself, or implicitly via references within the methods invoked by the parallel statement.)

Two SLOOP parallel statements therefore cannot **interfere** with each other, because each parallel statement executes as an **atomic** unit. **By grouping those actions that should be performed atomically into a single parallel statement, the correctness reasoning about interference is simplified.**

When a SLOOP program is mapped to a target architecture during the implementation phase, one has to ensure that the **mapped** parallel statement still executes as an atomic unit in order to preserve the correctness properties specified during the design phase. Possible ways to achieve this are discussed in Chapter 8.

The way in which reasoning about deadlock freedom is simplified in the SLOOP method is discussed in the next section.

Expressive power and **atomicity** are not the only characteristics of SLOOP statements that simplify correctness reasoning. The fact that all **parallel statements** are **always enabled** also makes it easier to reason about the program, since there is no need to consider the possibility that a parallel statement could be disabled when it is selected for execution. The correctness arguments during the design phase can be based on the guarantee that each parallel statement is enabled at all times and is executed infinitely often.

As in UNITY, another factor which simplifies correctness reasoning is the fact that it is **not** necessary to consider **flow of control** [ChMi88]. **When reasoning about the correctness of a SLOOP program, it is not important in what sequence the parallel statements of the various objects are executed; only the fact that they are executed infinitely often needs to be taken into account in correctness arguments.**

In Chapter 7 more details will be given regarding the SLOOP approach towards correctness reasoning.

4.3.6.5 Prevention of deadlock

A conventional concurrent object-oriented program has multiple threads of control. Each thread of control can be represented by a process. In Smalltalk-80, the process itself is implemented as an instance of the Process class [GoRo89]. The objects within the program execute within a specific thread of control (process) if their methods are invoked within the specified process. In all further discussions the term "process" refers to a thread of control. A process may or may not share the processor where it executes with other processes.

When processes are granted exclusive access to resources, deadlock can arise [Tane92]. A resource is anything that can only be used by a single process at a time. An object can therefore be viewed as a resource.

The four conditions that need to be present for deadlock to be possible are the following [Tane92, CES71]:

- **Mutual exclusion**. Each resource is either assigned to exactly one process or it is available.
- **Hold and wait condition**. Processes currently holding resources granted earlier can request new resources.
- **No preemption condition**. Once a resource has been granted to a process, the resource cannot be taken away from the process holding it; the process has to release the resource of its own accord.

- **Circular wait condition**. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next process in the chain.

If the objects in a program are viewed as the resources, deadlock is possible if the above conditions apply in the following way:

- While an object is busy executing a method which was invoked from within a particular process, the object will not execute the same or another method invoked by any other process. Thus, the object is a resource to which **mutual exclusion** applies.
- While an object is executing a method within a process, it might be necessary to send messages to other objects from within the current method. These other objects could be at one or more other processes. This implies that exclusive access to new objects could be requested while the process already has exclusive access to other objects (the **hold and wait** condition).
- Once an object is assigned to a process, the object remains assigned to it until the process releases it (when the execution of the relevant method completes). The object cannot be taken away from the process (the **no preemption** condition).
- Two or more processes must be involved in a circular chain, each requiring exclusive access to an object held by the next process in the chain (the **circular wait** condition). For example, suppose object X_o running in process X_p has sent a message to object Y_o running in process Y_p. Process X_p blocks while object X_o waits for a response. In the meantime, object Y_o has sent a message to object X_o and process Y_p blocks while object Y_o waits for a response. Since both processes are blocking, neither object can service the message received from the other. The two processes are therefore in a deadlock situation.

All four conditions must be true for deadlock to occur. If one of the conditions is absent, deadlock is not possible.

Deadlock prevention at the design level

In a SLOOP program there is no concept of processes at the design level. There is therefore no concept of a process which can be blocked while waiting for a response from another process. That is an implementation issue. During the design phase the behaviour of the system is described in terms of parallel statements. Since the semantics of SLOOP parallel statements are defined such that two parallel statements that share objects do not execute simultaneously and each parallel statement executes atomically, there can be no deadlock due to the mutual exclusion applying to the shared objects[24].

However, since SLOOP parallel statements may have conditions associated with them, it is conceivable that a program could consist of a set of parallel statements, each depending on a condition that is only set to true by another parallel statement in the set. Theoretically, one could therefore have the following scenario (each variable is initialised to zero):

```
  a := a + 1      if c > 0          "Statement S1"
[] b := b + 1     if a > 0          "Statement S2"
[] c := c + 1     if b > 0          "Statement S3"
```

Although each statement can execute when it is selected for execution, none of the executions will result in any state changes. In principle, one could therefore view an implementation where each of the above statements is assigned to a separate processor to be deadlocked (none of these variables can be set to a positive value by the environment).

In the SLOOP method, correctness reasoning about the parallel statements takes place during the design phase. As part of the correctness reasoning about liveness properties, one has to show that the conditions associated with the parallel statements that facilitate progress will eventually

[24] The definition of a shared object was given in the previous section (Section 4.3.6.4).

become true (if any of the relevant parallel statements are conditional). When reasoning about a safety property that specifies the complement of a liveness property, one has to show that no scenario exists which will result in the conditions never becoming true. It is clear that in **informal** correctness arguments it is far easier to reason about the liveness property than the safety property, because the liveness property is existential, whereas the safety property is universal. Thus, for a liveness property it must be shown that statements **exist** that will result in the required progress, whereas for the complementary safety property **all** the statements have to be inspected to check that none of them will prevent this progress.

The specification of a liveness property is usually also less complex than the specification of the complementary liveness property. This is because the safety property has to identify all the conditions that need to hold before the relevant progress will be prevented.

For example, one of the liveness properties that could be specified as a requirement for the above program is the following:

```
true leads-to a > 0                          "Property P1"
```

The complementary safety property is the following:

```
invariant   ¬(a ≤ 0 ∧ b ≤ 0 ∧ c ≤ 0)         "Property P2"
```

Informal correctness reasoning about liveness property *P1* entails showing that the variable `a` will eventually be greater than zero. This can be done by inspecting the initialisation statements as well as the parallel statements of the program (informal correctness reasoning procedures will be discussed in detail in Chapter 7). It is clear from the program text that there is only one parallel statement (statement *S1*) that will set the variable `a` to a value greater than zero (recall that the variable `a` is initialised to zero and statement *S1* is the only statement that modifies the value of `a`). However, this is a conditional statement, therefore one has to show that the condition associated with it (`c > 0`) will eventually become true.

The variable `c` (which is initialised to zero) is set to a positive value in statement *S3*, but only if the variable `b` is positive. One therefore has to show that `b` will eventually be set to a positive value. This is only done in statement *S2* (initially the value of the variable `b` is zero). However, statement *S2* is also conditional. Upon inspection of the condition associated with *S2*, it is discovered that it requires the variable `a` to be positive. Statement *S1* is the only one which modifies the variable `a`. However, the correctness argument started by trying to show that the condition associated with statement *S1* will eventually become true. The correctness arguments for liveness property *P1* therefore reveal a circular set of conditions in the parallel statements of the program. The program as presented above therefore does **not** satisfy property *P1*.

In the SLOOP method it is postulated that it will be possible to discover circular conditions in parallel statements via the correctness arguments of the liveness properties specified for the program. For this reason, the focus is on the specification of the appropriate **liveness** properties for SLOOP designs, rather than trying to identify all the relevant safety properties that would guarantee the absence of circular conditions in the parallel statements of a SLOOP program.

The remaining discussion of absence of deadlock properties therefore focuses on the implementation phase. As stated earlier in this section, deadlock related to the **mutual exclusion** of objects that are shared among the parallel statements of a SLOOP program is not at issue during the design phase. This is because the semantics of a SLOOP parallel statement dictate that each parallel statement is executed as an atomic unit and that parallel statements that share objects never execute simultaneously.

Deadlock prevention at the implementation level

The target architecture and the associated mapping of objects to processes have to be considered once the implementation phase is reached. In order to ensure that the correctness properties specified during the design phase are preserved during the implementation phase, the semantics of the SLOOP statements have to be preserved.

Firstly, one has to ensure that the atomicity of the parallel statements is preserved by the mapping. Thus, the mapped statement must always be able to complete its execution in a single atomic action even when it is sending messages to objects running in different processes. It should therefore be guaranteed that a situation cannot arise where the statements in the respective processes cannot complete execution due to deadlock.

For example, if the SLOOP program contains objects X_o and Y_o and their respective parallel methods contain parallel statements X_s and Y_s respectively, then one possible mapping would be to assign object X_o and its statement X_s to processor X_p and to assign object Y_o and its statement Y_s to processor Y_p. Note that statement X_s sends a message to object Y_o, resulting in the execution of the relevant sequential method of object Y_o if no deadlock occurs. Similarly, statement Y_s sends a message to object X_o, resulting in the execution of the relevant sequential method of object X_o if no deadlock occurs. Deadlock is prevented in the mapped program if one of the conditions for deadlock as described earlier can be removed. Let us consider each condition as a candidate for removal.

If the first condition (mutual exclusion) is removed, it implies that the execution of the statements of multiple sequential methods of the same object can be interleaved. The implications of allowing this are best explained via an example. (This is an artificial example created purely to demonstrate the principles.) Suppose object `X` has the following two methods:

```
message pattern limit: newLimit
method properties
"Total correctness"
true results-in methodReturnValue = self ∧ limit = newLimit
sequential
limit := newLimit                                          "Statement S1"
end-sequential

message pattern calculateNextElement
method properties
"Total correctness"
currentTotal < limit results-in
      methodReturnValue = self ∧
      nextElement notNil ∧
      currentTotal ≤ limit
sequential
  ^ self
      if currentTotal ≥ limit                              "Statement S2"
[] nextElement := self calculateNextElementValue           "Statement S3"
[] currentTotal := currentTotal + 1                        "Statement S4"
end-sequential
```

It is evident from the text of the `calculateNextElement` method that the method returns without executing any further statements if the instance variable `currentTotal` of object `X` is greater than or equal to `limit`. However, if `currentTotal` is less than `limit`, object `X` first calculates what the value of the next element should be, assigns it to the instance variable `nextElement` and then increments the `currentTotal` instance variable.

Now if the execution of the statements of the `limit:` and `calculateNextElement` methods can be interleaved, then the following execution sequence would be possible if object `Y` sends the `calculateNextElement` message to object `X` and object `Z` sends the `limit: 5` message to object `X` at approximately the same time (suppose `currentTotal` is equal to 5 and `limit` is equal to 6 at the start of the execution): *S2*, *S3*, *S1*, *S4*.

The above execution sequence implies that when statement *S4* is executed, it increments `currentTotal` to 6, which is greater than the new value of `limit`. This violates the total correctness property of the `calculateNextElement` method, which specifies that `currentTotal` should be less than or equal to `limit` after the execution of the method.

The interleaving of the execution of sequential method statements is therefore not allowed in the SLOOP method, because then it becomes difficult to reason about the correctness of such a method due to the interference by other methods. As stated in the previous section, a sequential method is always executed atomically; the execution of its statements is not interleaved with the execution of any other statements. The option of removing the mutual exclusion condition for deadlock is therefore not used in the SLOOP method.

One way of removing the second condition for deadlock (the hold and wait condition) is by disallowing the client object to block while waiting for the result of a message. However, this implies that synchronous messages are not supported. Synchronous message support is a requirement in the SLOOP method, since at the very least it is necessary to send accessing messages synchronously. That is to ensure that the conditions of a parallel statement can be checked and the associated actions executed in a single atomic step, thereby preventing any interference by other statements.

Another way of removing the hold and wait condition is by reserving all the required objects prior to the execution of a parallel statement. Thus, the statement may only start execution once all the objects required by the statement have been allocated to it. This guarantees that it will be able to complete its execution, since no other parallel statements may send messages to those allocated objects. This option has the additional advantage that it automatically also ensures that **parallel statements that share objects are not executed simultaneously**. (Recall that this was one of the requirements for the preservation of the semantics of the parallel statements during the implementation phase.) This is the option used in the SLOOP mappings to distributed architectures and it is discussed in more detail later in this section as well as in Chapter 8.

Removal of the third condition (the no preemption condition) implies that the implementation has to be able to take care of aborted messages. Thus, it has to be possible to restore the system to the state prior to the start of the execution of the current parallel statement. This could be very complex and is therefore not used in the SLOOP method.

The circular wait condition can be removed by allocating arbitrary numbers to the resources and requiring that resources only be requested in a specific (e.g. ascending) order. For example, if parallel statement X_s of object X_o sends messages to object Y_o, while parallel statement Y_s of object Y_o sends messages to object X_o, object X_o must always be reserved before object Y_o can be reserved (if lexicographical ordering is used). The following scenario illustrates why such ordering will prevent deadlock:

1. Parallel statement X_s at processor X_p is scheduled for execution.
2. Object X_o is reserved (since statement X_s belongs to object X_o).
3. Statement X_s starts executing.
4. Parallel statement Y_s at processor Y_p is scheduled for execution.
5. A reservation request is made for object X_o. Object Y_o is not yet reserved, since the request for object X_o has not yet been granted.
6. Object Y_o is reserved for parallel statement X_s.

7. Parallel statement X_s completes its execution and objects X_o and Y_o are released.
8. Object X_o is granted to statement Y_s.
9. Object Y_o is reserved for statement Y_s.
10. Statement Y_s starts executing.
11. Statement Y_s completes execution and objects X_o and Y_o are released.

Since resources can only be reserved in a specific order, there can be no cycles and therefore no deadlock.

The benefit of such a solution is that a process does not have to issue all its requests pertaining to a parallel statement prior to executing it. However, since there could be a multitude of objects and each parallel statement could have a different sequence in which messages are sent to these objects, it would be difficult to devise a satisfactory numbering scheme. For some statements the numbering scheme might result in the reservation of objects as they are required, whereas in other cases most (or all) of the objects would have to be reserved prior to the execution of the statement. For example, suppose a system consists of the following objects:

- an `Array` instance called `userConnections`,
- the `Connection` instances that are the elements of the `userConnections` array and
- an `OrderedCollection` instance called `inputQ`, which may also contain references to the `Connection` instances

Statement *S1* below needs to be executed:

```
inputQ addLast: ((userConnections at: i) serviceRequest)
      if (userConnections at: i) notNil                    "S1"
```

A lexicographic ordering based on the object names would result in the following reservation order for statement *S1*: (the `Connection` instances are ordered according to their positions in the `userConnections` array).

1. The `Connection` instance at position `i`,
2. the `inputQ` object and
3. the `userConnections` object.

However, when statement *S1* is executed, the `userConnections at: i` message expression in the *if* clause has to be evaluated first. Since the objects have to be reserved in a fixed order, it implies that for this specific statement all the relevant objects have to be reserved before it can start its execution. This is equivalent to the solution described earlier, where **all** the objects that are required by a parallel statement are reserved **prior** to the start of its execution. In the remainder of this section and in Chapter 8 it is assumed that a parallel statement only starts executing once **all** its required resources have been reserved. As stated earlier, this solution also ensures that parallel statements that share objects do not execute simultaneously.

Although the reservation of the required objects prior to the execution of a parallel statement ensures that each statement can execute to completion without any deadlock occurring, the steps that are executed to **acquire** the necessary objects (resources) can result in deadlock if a **distributed** implementation of the **object allocation algorithm** is used. Thus, if a central resource allocator assigns all the resources, then a single reservation request listing all the required resources for statement *S1* could be sent to the central resource allocator. The latter would then return a response granting the resources only if **all** the required resources can be allocated to statement *S1*.

However, if a distributed object allocation algorithm is used, then there is a resource allocator at each processor, where each resource allocator can only grant requests for resources at the local processor. This means that some resources at processor Y_p could be allocated to parallel

statement Y_s, while the request for resources at processor X_p also required by statement Y_s could still be outstanding. Similarly, some resources a processor X_p could be allocated to parallel statement X_s, while the request for resources at processor Y_p also required by statement X_s could still be outstanding. If parallel statements X_s and Y_s share the resources at processors X_p and Y_p, deadlock will result.

This problem can be overcome by ordering the requests to the various processors (i.e. by removing the circular wait condition). Thus, if parallel statement Y_s requires resources at processors X_p and Y_p, then a request is first sent to processor X_p requesting all the resources located at processor X_p that are required by statement Y_s. The resource allocator at processor X_p sends a single message to the resource allocator at processor Y_p when all the resources required at processor X_p can be granted. Only once the resources at processor X_p have been granted can the resources at processor Y_p that are required by statement Y_s be requested. The same algorithm is executed to reserve resources for parallel statements at other processors (i.e. the resources are always requested in the same order), which ensures that deadlock is prevented. The resource reservation algorithm is discussed in detail in Chapter 8.

Only those objects that are referred to as target objects in SLOOP statements need to be reserved. For example, if a statement contains the message expression
`inputQ addLast: aServiceRequest`
then it is only necessary to reserve `inputQ`. The `aServiceRequest` object is not used as a target object in this case (i.e. no message is being sent to it in this expression), so there is no need to reserve it.

In some cases a leaf parallel statement can only be executed if a condition at a higher level (non-leaf) parallel statement is satisfied. All conditions at higher levels are always anded to the conditions of the leaf parallel statements and all target objects referred to in those conditions must also be reserved before the leaf parallel statement can be executed. This is illustratred by Figure 4-7.

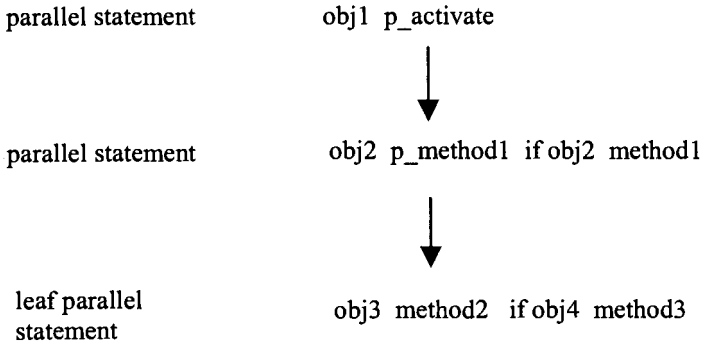

Figure 4-7. The role of *if* clauses in the parallel statement hierarchy.

Cyclic invocation of methods, where these methods execute as a result of the execution of the same parallel statement, does not result in deadlock. For example, if object A sends message `B1` to object B, and object B then sends message `A2` to object A, all as a result of the execution of the same parallel statement, no deadlock occurs. This is because **an object is allowed to execute more than one of its sequential methods if they are invoked via the same parallel statement**[25]. In such a situation there is no arbitrary interleaving of statements within the sequential methods of the object. Note that total correctness properties are specified for all

[25] This is also true for recursive calls of the same method.

sequential methods. This implies that the software designer is obliged to make sure that the termination conditions for each method will eventually be reached when the SLOOP statements of these methods are designed. Thus, if methods are invoked recursively or if messages are sent to the same object instance as a result of the execution of a particular parallel statement, the software designer has to take this into account when reasoning about the total correctness of these sequential methods. The software designer has to prove (informally) that each sequential method will eventually terminate[26].

For example, in Figure 4-8(a) below, parallel statement *p_A1* invokes sequential method `A1`. The latter contains statements *A1_S1*, *A1_S2* and *A1_S3*, where statement *A1_S2* invokes method `B1` of object B. In turn, statement *B1_S1* invokes method `A2` of object A, which contains statements *A2_S1* and *A2_S2*. When parallel statement *p_A1* is executed, the execution sequence will always be *A1_S1*, *A1_S2*, *B1_S1*, *A2_S1*, *A2_S2* and *A1_S3*. Since the execution sequence of the sequential statements is not arbitrary, the requirements of the SLOOP method are satisfied. The correctness properties of method `A1` take into account that method `A2` will be executed before statement *A1_S3* is executed.

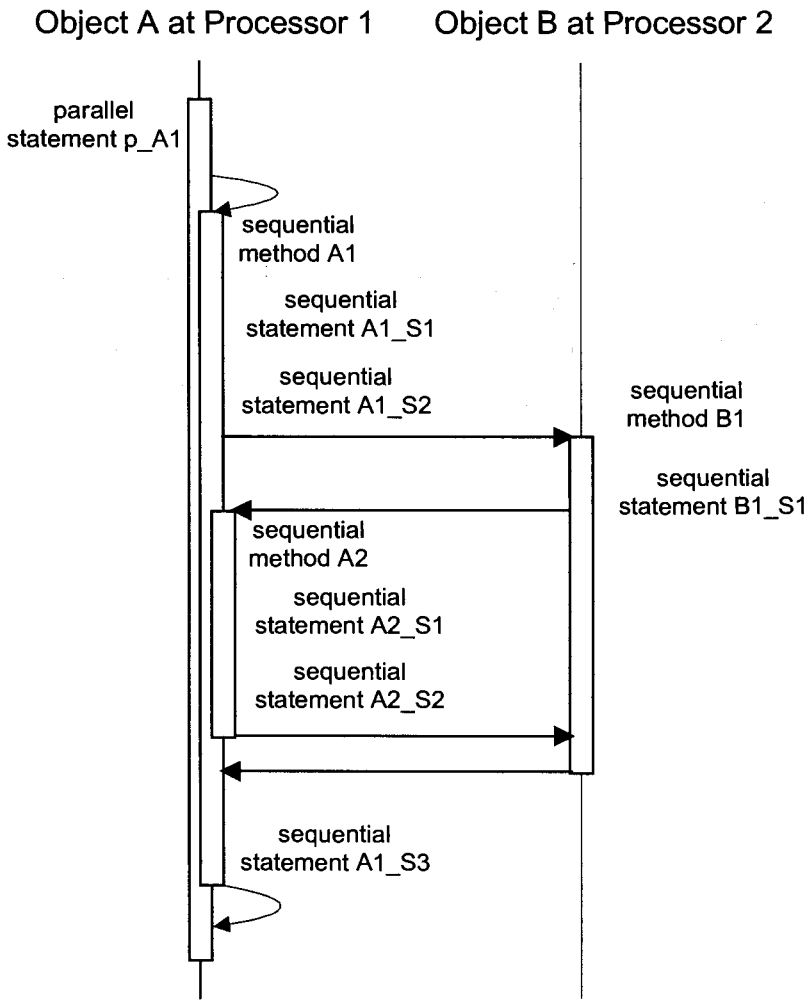

Figure 4-8(a). Cyclic invocation of the sequential methods of an object.

[26] In the case of object modelling with statecharts, as described in [HaGe97], a cycle of invocations that leads back to the same object instance is illegal, and an attempt to execute it will abort.

In contrast, the scenario shown in Figure 4-8(b) is **not allowed**. In this case the two methods of object A, viz. methods `A3` and `A2`, are invoked as a result of the concurrent execution of the two parallel statements *p_A2* and *p_B1*. Since the two parallel statements execute independently, statement *A2_S1* could be executed before statement *A3_S1*, after statement *A3_S1*, after statement *A3_S2* or after statement *A3_S3*. Thus, interference is possible. The correctness properties of method `A3` do not take the execution of method `A2` into account, since method `A2` is not invoked as a result of the execution of method `A3`.

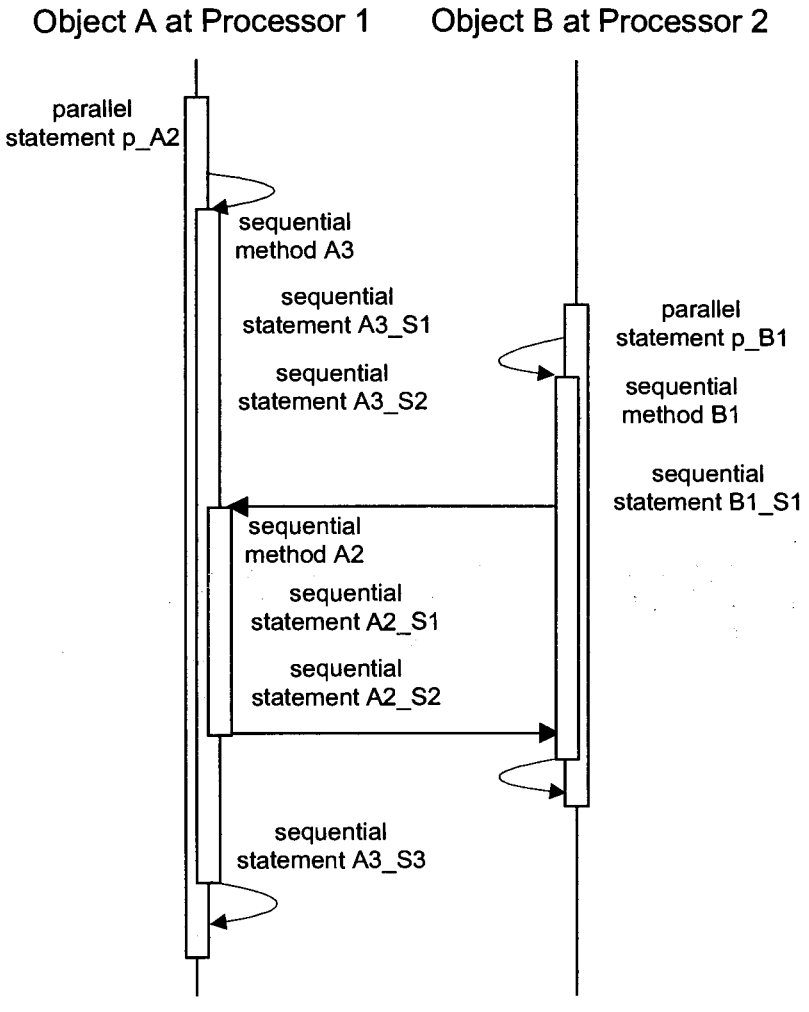

Figure 4-8(b). Example of an invalid execution sequence.

Note that the reservation of objects is only necessary if the SLOOP program is mapped to an architecture that comprises multiple processors. For example, if it is mapped to a program that comprises concurrent processes sharing a **single processor**, the execution of the parallel statements assigned to the various processes can be interleaved in any arbitrary way, but only one mapped parallel statement is executed at a time if each mapped statement executes to **completion** before the next statement is executed. In order to achieve this, one has to take care that messages that would relinquish control (such as the Smalltalk-80 `yield` message) do not form part of the mapped statements themselves. (Such messages could be used between the execution of the mapped statements to relinquish control to the process scheduler in order to enable the latter to schedule a different process.)

The mapping of SLOOP programs to various architectures is described in detail in Chapter 8. The latter also contains a further discussion of deadlock prevention during the implementation phase. Inter alia, it describes the procedures for identifying the objects that need to be reserved for the execution of a particular parallel statement.

It is clear from the above that the SLOOP method facilitates **simplified reasoning about deadlock**. At the design level the detail about object allocations to processes is not considered at all. The method therefore facilitates design at a **high level of abstraction**. If multiple resources are required for a specific action, then that requirement is indicated within a single parallel statement. The parallel statement construct therefore provides a high-level **encapsulation** mechanism for the low level resource allocation that is performed at the implementation level.

This is elucidated by the following example. In Dijkstra's well-known "dining philosophers" problem [Meye97, Dijk78], a philosopher has to obtain both the left and right forks before being able to eat. If only one fork is obtained at a time, a deadlock situation can arise where each philosopher in the circle has one fork and is waiting for another. However, if a philosopher obtains either both forks or none in a single atomic step, then there is no possibility of deadlock. In the SLOOP method, the software designer is provided with the necessary constructs to specify that the acquisition of forks is a single atomic action[27]. The design is therefore at a high level of abstraction, leaving the details of how such atomicity can be implemented to the implementation phase.

Absence of deadlock has to be considered when the SLOOP program is mapped to the target architecture during the implementation phase, but this is done in a **reusable** way. This aspect is described fully in Chapter 8. By preserving the semantics of the parallel statement during the implementation phase, the correctness properties associated with the SLOOP program are preserved for the implementation.

4.3.7 The SLOOP method and inheritance, encapsulation and polymorphism

No special constructs are required to support **inheritance** in the SLOOP method. As in conventional object-oriented languages such as Smalltalk-80 and C++, methods may be added or overridden in subclasses. This is true for both sequential and parallel methods. The sequential and parallel methods form part of the objects, i.e. SLOOP objects take full advantage of **encapsulation**. Note that this differs from the approach taken in the DisCo language [Kata-Web]. In the latter there is no concept of a method. Instead, objects participate in joint actions. These actions do not form part of the DisCo objects.

Polymorphism is supported in the SLOOP method as in conventional object-oriented languages. In Section 4.3.4.1 it was explained why the inclusion of message expressions in the syntax of correctness properties is required for full support of polymorphism.

As is evident from the above, the SLOOP method is a fully-fledged object-oriented method which provides all the advantages of inheritance, encapsulation and polymorphism.

4.4 Seamless analysis, design and implementation

It is clear from the above that the notion of a parallel statement executing infinitely often captures the essence of the SLOOP approach. In order to arrive at a specification which describes the dynamic behaviour of a system via a set of parallel statements, the requirements analysis and design have to be performed from this perspective.

[27] This solution does not guarantee absence of individual starvation, but the latter is another correctness property which is treated separately.

4.4.1 The requirements analysis phase

The requirements analysis phase is concerned with the representation of the **relationships** and **behaviour** of the objects in the problem domain. Artifacts that are created purely to facilitate the implementation of the solution, e.g. linked lists and arrays, are not mentioned at this stage. However, if a data structure models an abstract concept in the problem domain, it is perfectly valid to include such a structure at the analysis stage. For example, if a first-come, first-served ordering has to be modelled, it is appropriate to use a FIFO queue to model such a requirement.

It is important to note that abstraction during this phase does not imply being selective as far as the objects in the problem domain are concerned [RBPEL91]. For example, the objects that are described include the ones that participate in the normal behaviour of the system as well as those that are involved under abnormal conditions. Abstraction is achieved by avoiding design details such as the choice of data structures. The advantage of specifying all the objects in the problem domain during this phase is that it is useful to be aware of all the requirements of the system when having to choose between several **design alternatives**. It also enables one to determine whether an **appropriate framework** exists in the repository.

An object model is constructed to reflect the static structure of the system **and** its environment. The UML notation [RSC-Web] is used in graphical representations of the static structure. The next step is to determine the system boundaries, i.e. the classes that comprise the system are identified and their interfaces with the environment are specified.

Instead of constructing a dynamic and/or functional model, correctness properties are formulated to describe the behaviour of the system with respect to its environment. This encourages the paradigm shift from designing in terms of flow of control towards designing in terms of the properties of the system. At this stage the desired properties (i.e. correctness requirements) of the system are described informally. By systematically considering the various types of correctness properties oversights and deficiencies in the informal description of the system requirements are often revealed, as will be demonstrated in Chapter 5. Multiple iterations of the steps in the analysis phase may be required.

The target architecture, i.e. whether the system will consist of a single or multiple threads of control, is not considered during the analysis phase. Figure 4-9 provides a graphical representation of the steps followed during the analysis phase.

4.4.2 The design phase

The first step during the design phase is to search for a framework that represents the specified system. A match is only found if the properties advertised by the framework in the repository are applicable to the problem at hand. If it is a very large system, it is possible that one or more frameworks may be found, each representing one of the subsystems.

If a framework is found that represents the complete system, it is instantiated. If a mapping to an executable program is required at this level of refinement (for example, for prototyping purposes or if this is the final level of refinement), the mapping is performed for the SLOOP program that results from the instantiation of the framework. More details regarding the mappings are given in the next section, where the implementation phase is discussed. If the complete problem is solved, no further work is required, as shown in Figure 4-10(a). If not, the requirements at the next level of refinement are identified and the steps as shown in Figure 4-10(b) are applied to the remaining part of the problem.

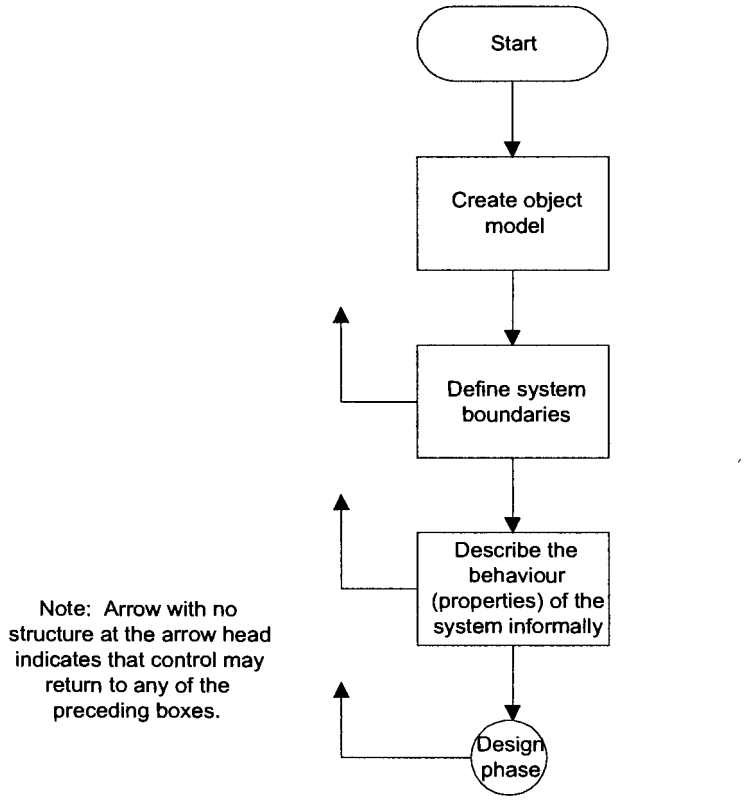

Figure 4-9. Requirements analysis phase steps.

If the initial repository search does not yield a framework representing the entire system, it may contain one or more framework(s) representing one or more subsystem(s). If such frameworks are found, they are instantiated. For the remaining subsystem(s), it is necessary to map the objects in the problem domain onto objects in the solution domain. Classes such as linked lists and arrays are now described. Additional properties may be identified, first informally and then more rigorously.

Once the interfaces of these classes have been identified, the repository is searched again, since an appropriate class, framework or design pattern pertaining to the newly identified classes might already exist. Finding a design pattern might result in some adjustments to the design of the system. If a framework is found, it is instantiated. All the resulting classes are incorporated into a SLOOP program that reflects the behaviour of the system at that level of refinement. Informal correctness arguments are used to check that the SLOOP program statements correctly represent the behaviour of the system as specified by the correctness properties.

The SLOOP program may be mapped to an executable program. If this is not the final level of refinement, the requirements of the next level are specified and the process is repeated.

The repository can be implemented as a database containing SLOOP programs. This takes advantage of the powerful search capabilities of a database management system. It also assists the designer in ensuring that classes are named uniquely when new systems are designed, thereby ensuring that there can be no confusion when referring to a particular class.

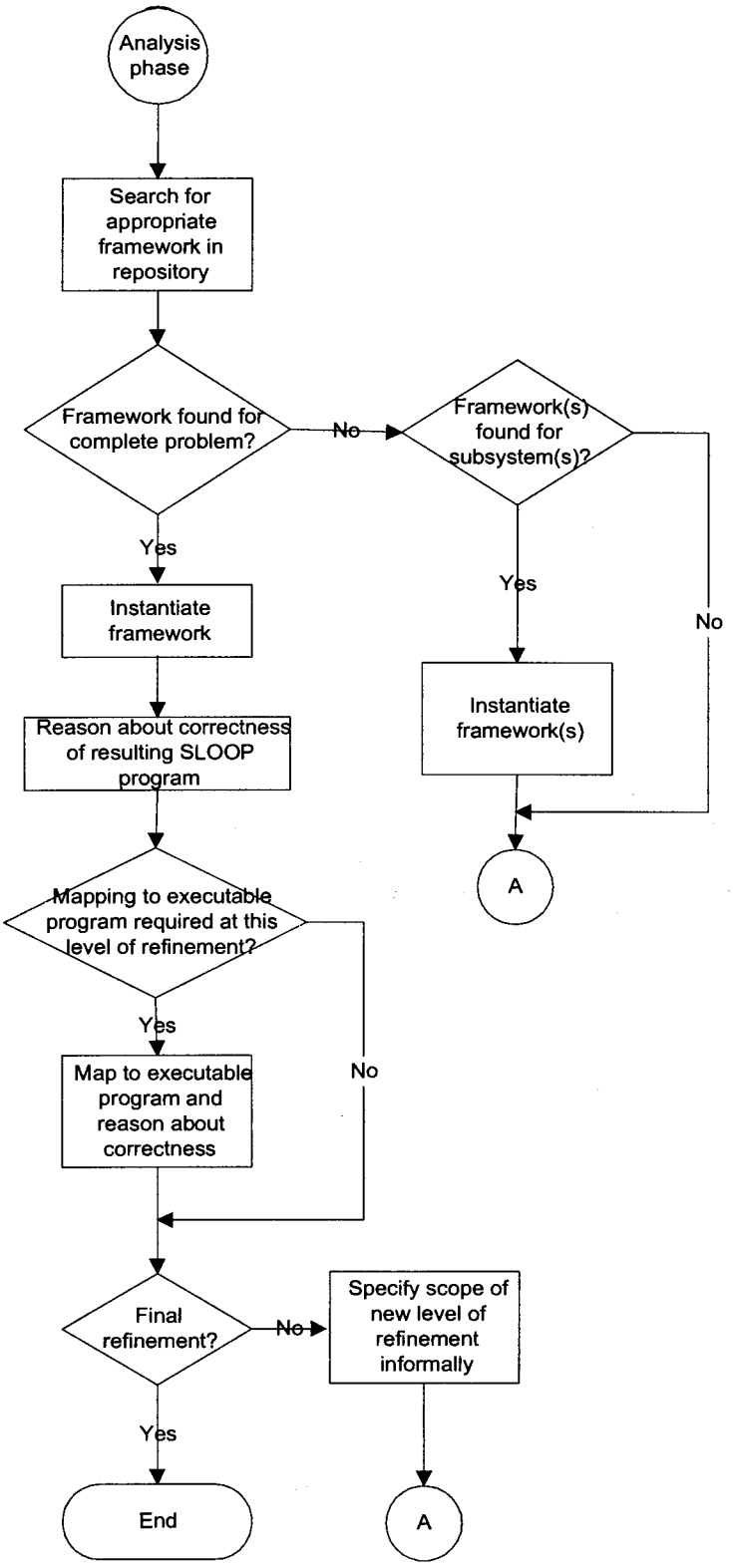

Figure 4-10(a). Design and implementation phases (Part 1 of 2).

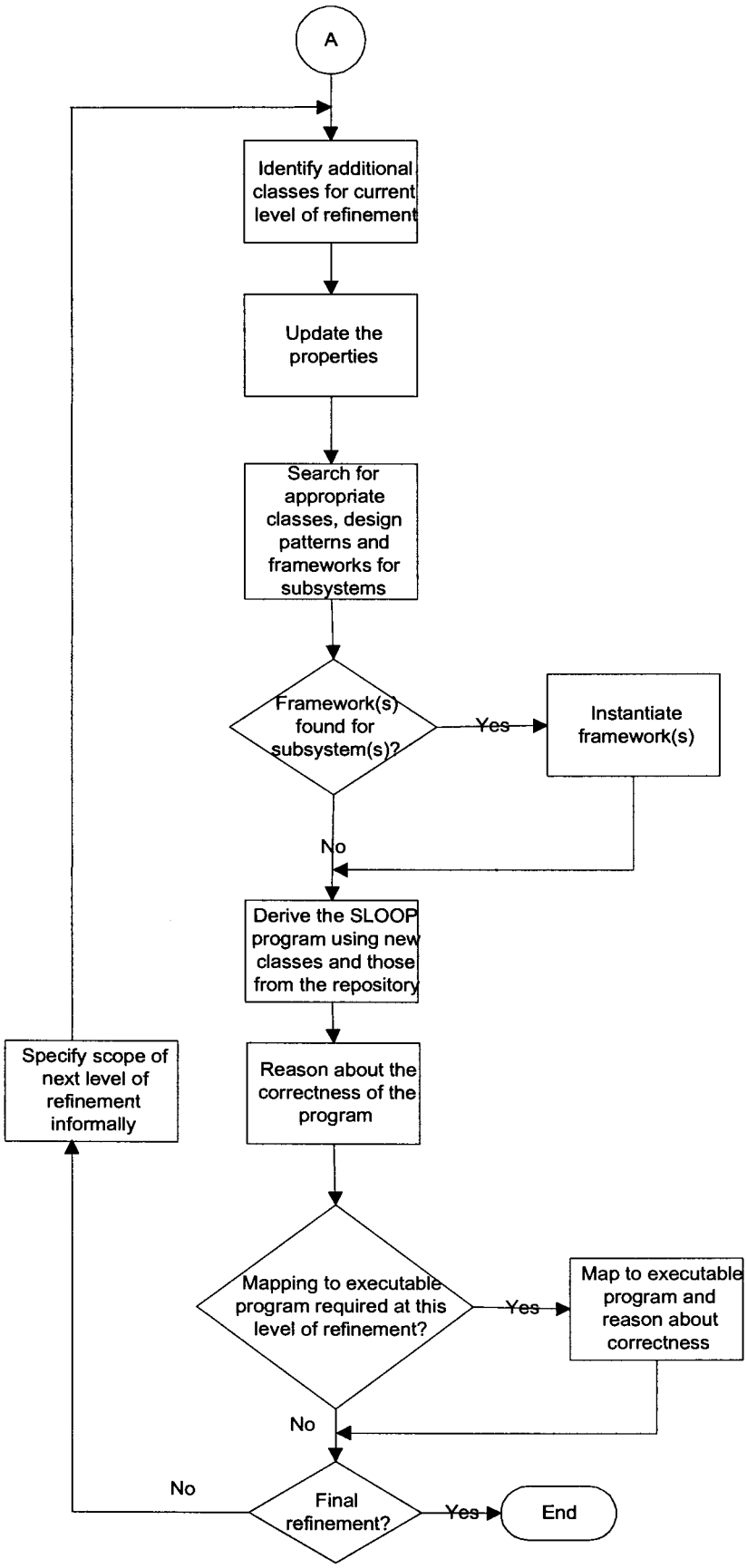

Figure 4-10(b). Design and implementation phases (Part 2 of 2).

Both the analysis and design phases are iterative processes. The problem specification may be revisited several times, since the properties of the classes/frameworks that have been selected from the repository may highlight aspects of the requirements that have been overlooked during previous iterations. Throughout the analysis and design phases the complexity of the system is managed by structuring it into hierarchies, layering it and/or partitioning it.

The deliverable of the design phase is a SLOOP program which satisfies the properties that have been specified. The SLOOP program models the behaviour of the system. This differs from the approach taken by Holzman in [Holz91], where the behaviour of a system is first modelled via extended state machines and then the correctness labels and assertions are **added**. The properties that are considered are divided into different categories, based on the cost involved to perform validation of the various properties. In contrast, a SLOOP program **results** from the specification of the relevant safety and liveness properties, i.e. the **"constructive approach"** towards software development is followed, as was indicated in the earlier chapters.

The allowable state transitions are reflected in the statements of the SLOOP program. The statements provide a description of the **object interactions** that does not merely represent a single scenario, but **all possible behaviours** of the objects. This was identified as a desirable feature in Chapter 3, Section 3.4.

The design may be refined repeatedly, revealing more objects and properties at each level of abstraction, taking care not to invalidate properties identified during earlier phases of refinement. **Design patterns** may be used from the start or they may be incorporated at a later stage. The SLOOP method follows the mixed specification approach: the program is not derived at the end of all the refinements of the properties. At each level of refinement a SLOOP program is derived and both the properties and the program are refined.

Once the program statements have been specified, informal arguments are used to reason about the correctness properties of the system. The main characteristic of these informal correctness arguments is the fact that the properties of each system are not proved from first principles. Each class has contractual **obligations** and the results are **reused** together with the class itself. These issues are discussed in more detail in Chapter 7.

When reusing an existing class, it is therefore only necessary to prove that the preconditions of the methods being reused are met; the proof of the postconditions are reused along with the class itself. In Chapter 8 it is shown how the reflective[28] facilities of Smalltalk can be used to check pre- and postconditions when a SLOOP program is mapped to an executable program.

Even during the design phase the target architecture is not considered. That is only done when the SLOOP program is mapped to an executable program.

4.4.3 The implementation phase

The SLOOP program may be mapped to an executable program at the completion of the design phase or prior to that if prototyping is required. This section serves as an introduction to the topic of SLOOP program mappings. The general approach is illustrated by an example. The heuristics for such mappings to **various architectures** are discussed in detail in Chapter 8.

The following SLOOP program was first introduced in Section 4.3.1. To recapitulate: it represents a system which simulates call centre behaviour, i.e. a system which accepts calls from service users and enqueues the associated service requests until they can be assigned to the

[28] The concept of computational reflection was described briefly in Chapter 1, Section 1.3.4 and will be discussed further in Chapter 8, Section 8.6.

relevant service providers. In order to simulate the behaviour of service users and providers, the TimerServices class is used to start random timers. The expiry of a timer results in an event which represents a new service request or an event which indicates that a service provider has finished dealing with a service request and is ready to handle the next one.

4.4.3.1 Mapping the activation-section

The excerpt of the SLOOP program below shows the *activation-section* of the `CallCentreSimulation` program. The remainder of the program comprises the `CC_ActivationPkg`, the `CC_CorePkg`, the `SystemUtilitiesPkg`, the `CC_SimulationInterfacesPkg` and the `SmalltalkLibPkg`. For the sake of brevity, the `TimerServices` class and those parts of the `CC_SimulationActivation` class that pertain to the `TimerServices` class are the only classes that are discussed in this example.

```
program CallCentreSimulation
      sequential
      aCCSimulationActivation :=
            CC_ActivationPkg:::CC_SimulationActivation setup
      end-sequential
      parallel
      aCCSimulationActivation p_activate
      end-parallel

package CC_ActivationPkg
class CC_SimulationActivation
      "Remainder of SLOOP description of CC_SimulationActivation
      class"
      ...

      "SLOOP descriptions of other classes in the CC_ActivationPkg"
      ...
end-package

      "Other packages"
      ...
end-program
```

The following is one possible Smalltalk-80 program derived from the above:

```
| aCC_SimulationActivation |

aCC_SimulationActivation :=
      CC_SimulationActivation setup.
[true] whileTrue: [aCC_SimulationActivation p_activate]
```

The *activation-section* is mapped to a set of Smalltalk statements in an Objectworks \ Smalltalk workspace [Parc90].

The mapping of the sequential statements is straightforward. The SLOOP statements are merely converted into Smalltalk statements. In the above example a temporary variable called `aCC_SimulationActivation` is created. It is used to store the new instance of `CC_SimulationActivation`.

The parallel statements are mapped by enclosing them in an infinite loop. In this example the `p_activate` message is sent to the newly created instance of `CC_SimulationActivation` within the loop.

The above example illustrates that not all classes need to be instantiated directly via a sequential statement in the *activation-section* of the program, even though it may contain parallel methods that need to be selected. For example, the `TimerServices` instance is instantiated via the instance creation and initialization methods of the `CC_SimulationActivation` class. Its parallel methods are activated via the `p_activate` method of the same class.

4.4.3.2 Mapping a package

A **class category** is used to group a set of related classes in the Objectworks \ Smalltalk [Parc90] environment. Such a class category may contain one or more packages and vice versa. In the above example the `CC_ActivationPkg`, the `CC_CorePkg` and the `CC_SimulationInterfacesPkg` are mapped to the `CC-Activation`, `CC-Core` and `CC-SimulationInterfaces` class categories respectively. All the classes within these packages are created within the respective class categories. The `SystemUtilitiesPkg` is mapped to the `SystemUtilities` class category. The `TimerServices` and other system utilities classes are created within this class category. The Smalltalk library classes are organised into multiple categories. All of these categories together comprise the `SmalltalkLibPkg`.

Note that the class categories are different from the method categories. The latter is used to categorise the methods of a class, e.g. into private, accessing, testing or modifying methods.

4.4.3.3 Mapping a class and its methods

The SLOOP superclass, class variables and instance variables are mapped directly onto their Smalltalk counterparts. The *macros-section* can be mapped in a number of ways, as described in Chapter 8. In this section the simplest mapping is described as an introduction to this topic, viz. each *macro-variable* is replaced with its corresponding *macro-expression* wherever it is used. This means that there are no references to the *macro-variables* in the Smalltalk mapping of a SLOOP program as shown in the example mapping of the `CC_Activation` class at the end of this section.

In Section 4.4.3.1 it was shown how the parallel statements within the *activation-section* of a SLOOP program are enclosed within an infinite loop when they are mapped to a Smalltalk program. A similar approach cannot be followed when a parallel method of a class is mapped, since that would imply that each parallel method would have to be executed within a separate process / thread of control (due to the fact that the infinite loop enclosing the parallel statements of each parallel method would prevent any other statements from executing within the same thread of control). In order to make it possible to execute multiple parallel methods within the same thread of control, each parallel method therefore has to return control to its client, which implies that the mapping of parallel method should not contain an infinite loop.

One possible solution is to execute **each statement** within a parallel method once during each invocation and to ensure that the method is **invoked** infinitely often (via the infinite loop in the *activation-section*). This mapping ensures that each statement in each leaf parallel method is executed **equally often** and infinitely often.

An alternative approach is to invoke each parallel method infinitely often, but to select only **one of the statements** within the parallel method during each invocation. In this case it has to be ensured that each statement within the parallel method is selected in turn. This mapping could result in some statements being executed more often than others, as illustrated by the following

example. Suppose p_method1 is a parallel method containing the following two parallel statements:

```
  objectX p_method2
[] objectY p_method3
```

If p_method2 contains only 1 parallel statement, while p_method3 contains 3 parallel statements, then the statement from p_method2 will be executed 3 times more often than any statement in p_method3.

However, since the requirement is only that each statement should be executed **infinitely often**, **not** infinitely often **and equally often**, the above mapping is acceptable.

Another consideration is the fact that the **level of atomicity** for a SLOOP program is the parallel statement in a leaf parallel method. In the second approach this is quite clear, since control returns to the *activation-section* after the execution of a single statement in a leaf parallel method. Reservation of objects starts afresh whenever control returns to the *activation-section*. Also, only those objects that are referenced as receivers in the **selected** statement at each nesting level need to be reserved during an execution of a parallel statement in the *activation-section*.

In the example below, it is shown how additional variables and statements are introduced in order to implement the second mapping. Since this information has nothing to do with the class itself, it is ideally implemented via the **reflective facilities** of Smalltalk. Thus, the Smalltalk-80 equivalent of the SLOOP statements appear in the Smalltalk base classes, while the additional variables and statements required in order to select the next statement for execution are implemented at the meta level. This will be discussed in more detail in Chapter 8.

SLOOP program fragments of the CC_Activation class and the corresponding Smalltalk-80 mapping are given next to illustrate the above concepts. (The derivation of these program fragments is fully discussed in Chapters 5, 6 and 8 and should be taken as given here.) The purpose of the CC_Activation class is to **activate** all the classes except the interface classes in the CallCentreSimulation program. The CC_SimulationActivation class is a descendant of the CC_Activation class and therefore inherits this functionality. In addition, the CC_SimulationActivation class activates the interface classes, i.e. the CommsProviderSimulator and ServiceProviderSimulator classes. For the sake of brevity, only those aspects related to activation of the TimerServices and Connection classes are shown below.

```
class CC_Activation
superclass SmalltalkLibPkg:::Object from SmalltalkLibRepository
instance variable names
      "The only variables relevant to the activation of the
      TimerServices and Connection classes are config,
      userConnections, timer and timerEventQ."
config
      "Refers to the object which handles the configuration of the
      system and is used by the TimerServices instance to obtain the
      maximum timeout value.  It is also used to obtain the maximum
      number of Connection instances supported by the system."
userConnections
      "This is an instance of the Array class which contains all the
      Connection class instances."
timer
      "The TimerServices instance."
timerEventQ
      "The TimerServices instance places a TimeoutElement instance in
      this queue when the corresponding timer has expired.  The
```

```
        TimerServices clients inspect this queue in order to determine
        whether a timer has expired."
...
class macros
maxConn ≡ config maximumConnections
        "Number of simultaneous user connections supported"
...
class properties
        invariant
        config notNil ∧
        userConnections notNil ∧
        < ∀ i where 1 ≤ i ≤ maxConn :: (userConnections at: i) notNil
        > ∧
        timer notNil ∧
        timerEventQ notNil ∧
        maxConn > 0 ∧
        ...

        "The CC_Activation class is an abstract class and should not be
        instantiated"
        invariant    <∀ anObject :: anObject class ~~ CC_Activation
                     >

instance methods
category private
        message pattern initialize
        method properties
        "Total correctness"
        true results-in
                methodReturnValue = self ∧
                config notNil ∧
                self postconditions: (#initManagement) ∧
                userConnections notNil ∧
                < ∀ i where 1≤i≤maxConn :: (userConnections at: i )
                notNil
                > ∧
                ... ∧
                timer notNil ∧
                (timer class) postconditions:(#setup:)
                withArguments: #(config) ∧
                timerEventQ notNil
        "Note that the receiver of the postconditions:withArguments:
        message is the expression (timer class) instead of
        SystemUtilitiesPkg:::TimerServices. This is done to facilitate
        subclassing without violating the correctness properties. If
        the actual class name had been used here, then the property
        would no longer have been valid if a subclass of TimerServices
        had been instantiated at this point. Recall that correctness
        properties must be preserved during subclassing."

        "At this stage (i.e. before the subclass has completed the
        execution of its instance creation method) the class invariants
        do not need to hold yet, so it should be stated explicitly that
        once the predicate self postconditions: (#initManagement) holds,
        it continues to hold. That is a requirement, since many of the
        subsequent statements in the method depend on it. The following
        correctness property specifies this requirement."
        stable config notNil ∧ self postconditions: (#initManagement)
```

```
        "Note that self postconditions: (#initManagement) implies that:
        maxConn > 0 ∧
        ... "

        sequential
        config := self initManagement
        [] userConnections := SmalltalkLibPkg:::Array new: maxConn
        [] < [] i where 1≤i≤maxConn :: userConnections at: i
              put: (self initConnection: i)
          >
        [] ...
        [] timer:= SystemUtilitiesPkg:::TimerServices setup: config
        [] timerEventQ := SmalltalkLibPkg:::OrderedCollection new
        end-sequential

        message pattern initManagement
        method properties
        "Total correctness"
        true results-in
              methodReturnValue notNil ∧
              (methodReturnValue class) postconditions:(#setup)
              "Again the explicit reference to a class name (in this
              case CC_CorePkg:::Configuration) is avoided in order to
              ensure that subclasses do not violate the correctness
              property."
        sequential
        ^CC_CorePkg:::Configuration setup
        end-sequential

        message pattern initConnection: index
        method properties
        "Total correctness"
        true results-in
              methodReturnValue notNil ∧
              (methodReturnValue class) postconditions:(#setup:)
              withArguments: #(index)
        "Again the explicit reference to a class name (in this case
        CC_CorePkg:::Connection) is avoided."
        sequential
        ^CC_CorePkg:::Connection setup: index
        end-sequential

category cyclic
        message pattern p_activate
        method properties
        "These are the properties of the system as identified during the
        analysis phase.  For brevity they are not listed, since they are
        not relevant to the current discussion."
        "The p_activate method is only invoked once the CC_Activation
        subclass has been instantiated.  The class invariants of the
        CC_Activation subclass that has been instantiated are therefore
        guaranteed to hold before the p_activate method is executed.
        Each statement executed by the p_activate method has to preserve
        these invariants."
        parallel
        ...
        [] timer p_runTimer: timerEventQ
        "This statement invokes the p_runTimer: method of the
        TimerServices class.  For easy reference, the functionality of
        that method is summarised here:  Whenever a timeout occurs, the
```

```
      TimeoutElement instance representing the timeout is added to the
      end of the timerEventQ, which indicates to the requestor that
      the specified timer has expired."

      [] < [] i where 1≤i≤maxConn :: self p_executeConnection:
            (userConnections at: i)
        >
      "This statement invokes the p_executeConnection method of the
      CC_Activation class for each instance of the Connection class.
      The purpose of the p_executeConnection: method is to invoke the
      parallel methods defined for the Connection instance.  The
      functionality of these methods is as follows:  When a connection
      has entered the 'TERMINATING' state, the communication provider
      agent is requested to terminate the connection.  Once all the
      procedures to terminate the connection have been completed, the
      connection and its associated service request are reset to their
      initial states."
      end-parallel
```

The Smalltalk mapping of the `CC_Activation` class is shown below. For the sake of brevity, it is assumed that the `p_activate` method contains only those statements that are relevant to the `TimerServices` and `Connection` classes, i.e. one statement for the `TimerServices` instance and one statement for each of the `Connection` instances.

```
class name   CC_Activation
superclass   Object
instance variable names
p_activateTally
      "Contains the number of statements in the p_activate method."
p_activateCycleIndex
      "This variable is used to calculate the next parallel statement
      to be selected for execution in the p_activate method.  It is
      incremented modulo p_activateTally."
config
userConnections
timer
timerEventQ

instance methods
private
      initialize
      "Note the mapping of the maxConn macro-variable in the
      statements below."

      p_activateTally := 1 + (config maximumConnections).
      "There is one parallel statement to invoke the p_runTimer:
      method of the TimerServices class and one parallel statement for
      each of the Connection instances."

      p_activateCycleIndex := p_activateTally - 1.
      config := self initManagement.
      userConnections := Array new:(config maximumConnections).

      1 to: (config maximumConnections) do:
      [:i| userConnections at: i put: (self initConnections: i)].
      "This demonstrates the mapping of a quantified-statement-list in
      a sequential method"
      ...
```

```
        timer := TimerServices setup: config.
        timerEventQ := OrderedCollection new

        initManagement
        ^Configuration setup

        initConnection: index
        ^Connection setup: index


cyclic
        p_activate
        "Again for simplicity only those statements related to the
        TimerServices and Connection instances are shown."

        p_activateCycleIndex :=
              ((p_activateCycleIndex + 1) \\ p_activateTally).
        "Determine which statement should be executed.  The '\\' symbol
        is the Smalltalk-80 modulo operator."

        (p_activateCycleIndex = 0)
        ifTrue:  [timer p_runTimer: timerEventQ]   "(statement 0)"
        ifFalse: [(p_activateCycleIndex > 0 and:
                  [p_activateCycleIndex ≤ (config maximumConnections)])
                  ifTrue: [self p_executeConnection:
                        (userConnections at: p_activateCycleIndex)]
                  "statements 1 to config maximumConnections"
                  "The part in italics demonstrates the mapping of a
                   quantified-statement-list in a parallel method."
                  ]
```

The `p_activateTally` and `p_activateCycleIndex` instance variables are introduced in order to select a statement to execute in the `p_activate` method. During initialization the `p_activateTally` variable is set to the number of parallel statements in the `p_activate` method. The `p_activateCycleIndex` variable is used to select the next statement from the `p_activate` method. It is therefore incremented modulo `p_activateTally`. Its values range from zero to `p_activateTally - 1`. Initially it is set to `p_activateTally - 1`. That ensures that the first execution of the `p_activate` method will result in the first statement being executed. In Smalltalk-80 there is no notion of a parallel and a sequential method. All the statements in the Smalltalk-80 mapping of a parallel method are executed whenever the method is invoked. It is by introducing the additional variables as illustrated above that selective execution is achieved.

In the `CC_Activation` example the Smalltalk mappings of *quantified-statement-list*s were shown for both sequential and parallel methods. In a sequential method (as was demonstrated in the `initialize` method) the list is simply mapped to a sequence of statements using the `to:do:` message. In the case of `p_activate` (the parallel method in this example), the statement that is executed depends on the value of `p_activateCycleIndex`. If the value of this variable is within the range of the quantification of the *quantified-statement-list*, then the appropriate statement is executed. The above program excerpt contains a very simple mapping. If a parallel method contains multiple *quantified-statement-list*s, the algorithm is adapted to determine the start and end values of each quantification and to execute the corresponding statements, as will be shown in Chapter 8. The latter also covers the mapping of statements with multiple *statement-component*s and *component-part*s.

The parallel method `p_runTimer:` contains statements that monitor the timers in the system and which append the corresponding `TimeoutElement` instances to the `timerEventQ` if they have

expired. The parallel method only has the desired effect if it is invoked infinitely often. There is no loop in the `p_runTimer:` method itself. The statements are executed repeatedly because the `p_runTimer:` method is invoked infinitely often (as a result of the infinite loop in the mapping of the parallel statements in the *activation-section*).

This section merely serves as an introduction to the topic of obtaining executable programs from SLOOP programs. Issues such as the various target architectures and levels of parallelism are discussed in Chapter 8. The role of the correctness properties during the implementation phase is also addressed in Chapter 8.

4.5 Summary

This chapter has provided an overview of the SLOOP method. It has shown how the SLOOP method encourages a software development approach **driven by correctness properties**. This was reflected in the steps listed for each software development phase. The purpose of such a "constructive approach" is to produce more reliable and functionally correct software.

Although the computational model is unconventional, the basic concepts of an object-oriented approach still apply. It was demonstrated how the concept of statements that execute infinitely often can be integrated with the concepts of classes and methods. It was shown how object-oriented features such as inheritance, polymorphism and encapsulation fit into the SLOOP method. The notation provides for the inclusion of correctness property specifications in the class definitions, which facilitates reuse of these properties. By ensuring that all these aspects of object-orientation are embraced, the foundation is laid for addressing the **scalability** problem. The latter is discussed further in the chapters to follow.

A major part of this chapter was devoted to the syntax and semantics of SLOOP programs. The purpose and meaning of each construct were described. In Section 4.3.4.4 the definitions of the basic logical relations used in in correctness properties were presented. These definitions provide the necessary notation for rigorous specifications of correctness properties. As evident from the steps followed during the SLOOP software development process, these specifications are used in **informal** correctness arguments. Thus, it is not necessary to be proficient in the mathematics required by formal methods in order to benefit from these specifications.

The section describing SLOOP statements was particularly significant because it dealt with some core aspects of the method. It showed that **concurrency** is modelled by the arbitrary interleaving of SLOOP parallel statements. It also models **asynchrony** because the execution of each parallel statement is independent of the execution of every other parallel statement in the system. However, a parallel statement may consist of multiple components, which facilitates the modelling of **synchrony**.

Since parallel statements execute infinitely often and because they may be conditional, they are used to model the **active** nature of an object. Thus, an event can be viewed as an occurrence which changes to true or false the condition associated with a parallel statement. The statement executes infinitely often, so the object will always eventually **react to the event**, provided the value of the *if-clause* associated with the parallel statement remains the same. The operation of a system is driven by these parallel statements and the conditions associated with them.

The **expressive power** of the SLOOP statement enables one to group all the actions that need to execute **atomically** into a single parallel statement. This is the mechanism that is provided to prevent interference and race conditions. This chapter has also shown that the problem of guaranteeing absence of deadlock is deferred to the implementation phase, where the reservation of objects is used as a deadlock prevention mechanism. This demonstrated another feature of the SLOOP method, i.e. the fact that the mapping to target architectures is an implementation issue.

The design applies to all types of architectures. Thus, the SLOOP parallel statements, used during the design phase, are at a high level of abstraction. In the SLOOP method the solution domain is therefore viewed as a collection of **interacting** objects. This applies regardless of whether the system comprises a single sequential process or multiple concurrent processes.

The last part of this chapter described the mechanism provided in order to achieve **seamless analysis, design and implementation**. During the analysis phase an **object model** is constructed to reflect the **static structure** of the system. However, the **dynamic behaviour** of the system is described by the **correctness properties** of the system. During the design phase the problem domain objects are mapped onto solution domain objects and the correctness properties are **refined** accordingly. Again the emphasis is on the correctness properties of the system. During the design phase **a SLOOP program is derived** at each level of refinement. The statements of the SLOOP program specify the **object collaboration**.

Finally, it was shown how a SLOOP program could be mapped to an executable Smalltalk program. Since the two languages are closely related, the mapping is straightforward, which facilitates easy **prototyping**. The fact that it is so easy to create executable programs, as well as the availability of the Smalltalk-80 class library, made it possible to experiment with the concepts incorporated into the SLOOP method without requiring a huge outlay in terms of developmental resources.

The next five chapters elaborate on various aspects of the SLOOP method. Chapter 5 covers the analysis phase. Chapters 6 and 7 discuss the design phase, with Chapter 7 being devoted to the aspect of informal correctness reasoning during this phase. Chapter 8 addresses the implementation issues and Chapter 9 shows how to incorporate design patterns into a SLOOP design.

CHAPTER 5

REQUIREMENTS ANALYSIS FROM THE SLOOP PERSPECTIVE

5.1 Introduction

When applying the SLOOP software development method, the requirements analysis phase has two important deliverables: the class diagram and a set of correctness properties describing the expected behaviour of the system, as well as a set of correctness properties for each individual class comprising the system. The class diagram is constructed by identifying classes and the relationships between them. The correctness properties are derived from the informal problem statement.

The purpose of the specification of the correctness properties is manifold:

- It encourages the designer to focus on correctness issues from the outset, i.e. this forms part of the **"constructive approach"** towards software development. As will be shown later on in this chapter, the designer is inclined to consider not only what the required behaviour should be, but also what it should not be. **Error conditions** are therefore identified at this early stage of system development.
- The identification of correctness properties encourages the software designer to perform a **careful analysis** of the informal problem statement. The checklist that is presented in Section 5.2.4 guides the designer to view the problem statement from many different perspectives and this exercise usually **reveals deficiencies** in the problem statement. These could be ambiguities, inconsistencies or omissions. The case study that is used in this chapter provides several examples of this aspect of the SLOOP method.
- Since the problem statement has to be studied from so many different perspectives, it aids the software designer in obtaining a **thorough understanding** of the requirements. A better understanding during the initial phases of software development is likely to result in fewer problems during the later phases.
- The specification of the behaviour of the system and its constituent classes via correctness properties aids the software designer in **identifying suitable frameworks and classes** in the repository of reusable artifacts during the design phase. The correctness properties of the system under development are compared with those of the reusable artifacts. It is not necessary to study the code of each reusable class in detail in order to understand its behaviour. It should suffice to study its correctness properties only.

In Chapter 1 it was stated that **scalability** needs to be addressed when devising a software development method. The case study that is used in this and other chapters was chosen specifically in order to illustrate this aspect of the SLOOP method. Especially in Section 5.4, it will be evident that the task of specifying correctness properties is not insignificant. Considerable time and effort is required in order to come up with meaningful properties. However, such an investment has the advantages as pointed out in the list above.

Furthermore, it will be noticed that the correctness properties listed in Section 5.4 are only specified **informally**. This is to avoid wasting time and effort translating informal properties into formal ones if this has already been done before, i.e. in the case where a similar system or parts of it can be found in a repository of reusable artifacts. Since the artifacts contained in a repository will have been implemented already, all aspects of their behaviour will have been specified in detail, both informally and formally.

The idea is therefore to specify enough correctness properties to capture all the requirements of the system under development. One of the aims of this step is to use the resulting specification to try and find **reusable** artifacts that might match these requirements in a repository. If inspection of the behaviour of those artifacts does not reveal any **conflicts** with the requirements of the system under development, it means that a great deal of **effort is saved**. This is because the formal specification of the behaviour of the matching artifacts can be reused in the new system. In addition, the refinements captured in the specification of these matching artifacts can also be reused if applicable. If a class of the system under development cannot be matched, its behaviour is specified formally during the design phase.

Understandability is another issue which receives attention in this chapter. A stated goal is to make the SLOOP method accessible to software engineers that are not necessarily trained in formal methods. This is one of the reasons for developing the **checklist** of correctness properties as presented in Section 5.2.4. It helps the software designer not to overlook some aspects of the behaviour of the system under development and is therefore an aid in avoiding **underspecification**.

Experience with the SLOOP method has shown that one of the difficult aspects of the specification of the properties at the analysis level is to ensure that the properties are formulated without implementation bias, i.e. the problem is to avoid **overspecification**. This topic is also covered in this chapter.

In Chapter 1 it was stated that the SLOOP method is a **"lightweight"** formal method. Thus, although no formal proofs are produced, the correctness properties have to be specified in a rigorous fashion. This is to ensure that these properties are **unambiguous** and **consistent**. Correctness arguments would not have much value if one could not at least rely on the unambiguity and consistency of the properties that are being reasoned about. During the requirements analysis phase the first steps towards achieving this are taken. The formalisation of the properties is performed during the design phase, when the final structure of the system has been determined (i.e. the classes that make up the system under development have been finalised after design level refinements and the incorporation of reusable frameworks, design patterns and/or classes).

It is not expected that the first version of the correctness properties will be the final one. The SLOOP method allows for multiple iterations of the steps of the requirements analysis phase and even iterations between the design and analysis phases. However, due to the fact that the designer is encouraged to analyse the problem statement very thoroughly during the analysis phase, the need to return to the analysis phase during the design phase tends to be reduced. The properties in the example shown in this chapter did **not** require iteration between the specification of analysis level properties and the development of SLOOP statements (a design phase activity).

During the analysis phase **multiple iterations** are usually caused by **deficiencies** in the initial informal problem statement and also as a result of the fact that the software designer gains **better insight** into the problem statement as he/she works through the checklist of correctness properties. Only the final iteration is shown in the example that is worked out in this chapter. However, wherever applicable the reasons for multiple iterations are pointed out.

The remainder of this chapter is organised as follows: First of all the correctness property definitions that were specified in terms of temporal logic in Chapter 2 are rewritten in terms of the SLOOP logical relations. This culminates in a checklist of useful correctness properties.

The definitions in this chapter are required in order to assist the software designer in formulating the correctness properties during the various development phases. Although the correctness properties are only formalised during the design phase, it is necessary to have a good understanding of what they are about when specifying them informally during the requirements analysis phase, hence the inclusion of their formal definitions in this chapter.

The latter part of this chapter demonstrates how the SLOOP method is applied during the requirements analysis phase. First of all the informal problem statement for the system that is used as running example in the remainder of this chapter as well as in the ensuing chapters, is presented. It is shown that the construction of the **class diagram** is an essential part of the requirements analysis phase, but any existing technique such as the one described in [RBPEL91] suffices to identify the classes and their relationships. The distinguishing aspect of the SLOOP method is its approach towards describing the **behaviour** of the system under development, which is the focus of the discussions in this chapter. In Section 5.4, the following issues are highlighted:

- It demonstrates how the different correctness property types are **interpreted** in terms of an informal problem statement.
- It shows how the specification of correctness properties can **reveal deficiencies** in an informal problem statement.
- It illustrates how the **"constructive approach"** results in a specification that does not only contain properties describing the desired behaviour of the system, but also properties describing the conditions that need to be satisfied in order to **prevent undesired behaviour**.
- It addresses the issue of **overspecification**.
- It discusses the need for **consolidation** after one has worked through the checklist of correctness properties, i.e. it highlights the need to check for and remove **inconsistencies** and **redundancies**.
- Finally, it is shown how the classes that make up the system under development have to **preserve** the correctness properties of the latter.

5.2 Useful correctness properties

In Chapter 2 it was stated that the correctness properties for concurrent systems are generally categorised as either safety or liveness properties. Manna and Pnueli [MaPn81a] define a third category, viz. precedence properties. A number of useful properties within these categories were then given, which serve as a basis for the **checklist** developed in this chapter.

In Chapter 2 the definitions of these properties were presented in terms of temporal logic. Location counters featured prominently in these definitions. The properties are now **rewritten** in terms of the logical relations used in the SLOOP method. Definitions of the UNITY logical relations **unless**, **ensures**, **leads-to**, **stable**, **invariant**, **detects**, **until** and **precedes** were given in Chapter 2, Section 2.5.5. Their SLOOP counterparts were described in Chapter 4, Section 4.3.4.4. The SLOOP **results-in** logical relation was also defined in the latter. All the SLOOP logical relations except the **results-in** logical relation are based on the SLOOP computational model.

Note that most of the definitions of the SLOOP logical relations refer only to the parallel statements of a SLOOP program. This is because the SLOOP parallel statements are the atomic units that are executed infinitely often, as required by the SLOOP computational model. (In the case of the **unless** logical relation used in the **class** *properties-section*, the definition also refers to

sequential methods, since the **safety** properties of a **class** must also be preserved by the sequential methods. However, the definition does not refer to individual sequential statements, since a sequential method is always executed as an atomic unit, usually as part of the execution of a parallel statement.)

As pointed out in Chapter 4, Section 4.3.4.3, the correctness of a sequential method is described via a total correctness property. The total correctness property of a sequential method is defined in terms of the **results-in** logical relation, which is based on the conventional model of control flow.

The following symbols are used in the definitions of the correctness properties:

$\varphi(\bar{x})$	The precondition that restricts the set of inputs $\bar{x}$ for which a program is supposed to be correct.
$\psi(\bar{x}, \bar{y})$	The statement of correctness, i.e. the relation that should hold between the input values $\bar{x}$ and the output values $\bar{y}$.
FP	Fixed point of the program, i.e. the execution of any parallel statement of the program does not change the state of the program.

In the case of UNITY, a fixed point of a program *G* is a predicate FP which is defined as:
$FP \equiv <\forall$ statements *s*: *s* in $G \wedge s$ is $X := E :: X = E>$
where *G* is a program, *X* is a variable and *E* is an expression [ChMi88]. Thus, if a fixed point exists, it is a program state such that the execution of any statement of program G leaves the state unchanged.

The above definition is not suitable for SLOOP programs, even if the phrase '∀ statements *s*' is replaced by '∀ **parallel** statements *s*' for the SLOOP case. This is because SLOOP statements are not necessarily of the form $X := E$. A SLOOP statement may also be made up of one or more message expressions only.

The SLOOP definition of a fixed point of a program *G* is therefore as follows:

$FP \equiv <\forall$ classes *C* **where** *C* in *G* ::
    <∀ parallel methods *PM* **where** *PM* in *C* ::
        <∀ statements *s* **where** *s* in *PM* :: $\alpha = \beta$
        >
    >
>

where α represents the program state prior to the execution of statement *s* and β represents the program state after the execution of statement *s*.

Thus, a fixed point of a SLOOP program leaves the program state unchanged after the execution of any parallel statement of any class in the SLOOP program. Note that a fixed point may or may not exist in a program.

One of the goals of the SLOOP method is follow a **unified** approach towards software development. Thus, the aim is to enable the software designer to use a single mechanism to specify any type of system, i.e. sequential, concurrent, terminating, non-terminating, etc. SLOOP programs are always non-terminating, since the parallel statements execute infinitely often. Terminating programs are viewed as a special case of a non-terminating program. More precisely, if a **stand-alone** SLOOP program G has reached a fixed point, then it does not matter

whether the execution continues or is terminated[1]. Correctness properties that are only meaningful for terminating **programs**, i.e. the partial and total correctness properties, are therefore formulated in terms of fixed points. (As noted above, the total correctness properties for sequential **methods** are defined in terms of the **results-in** logical relation.)

5.2.1 Safety properties

5.2.1.1 Partial correctness

Partial correctness is represented by the following relation:
$\varphi(\bar{x}) \wedge \neg$FP **unless** FP $\wedge\ \psi(\bar{x},\bar{y})$.

Thus, if a program reaches a fixed point, then the results of the program will be correct. The partial correctness property is only meaningful for a terminating program.

Since sequential methods are based on the conventional model of control flow, the partial correctness definition presented in Chapter 2, Section 2.4.4.1, can be used to describe the partial correctness of a sequential method:
$\varphi(\bar{x}) \supset\ \Box$ (at $\bar{l}_e \supset \psi(\bar{x},\bar{y})$)
where $\bar{l}_e$ is the vector of the exit locations of the sequential method. An exit location is one where the statement is preceded by the caret (^) symbol, or otherwise it is the last statement of the sequential method.

Note however that a SLOOP sequential method always returns a value, which is denoted by the special SLOOP variable `methodReturnValue` in the method correctness properties. As a result, it is not necessary to refer to explicit exit locations in the partial correctness properties of sequential methods. It suffices to refer to the required value of `methodReturnValue` in order to indicate that an exit location has been reached. The following definition of partial correctness is therefore used for SLOOP sequential methods:
$\varphi(\bar{x}) \supset\ \Box$ ((methodReturnValue = ρ) $\supset \psi(\bar{x},\bar{y})$)
where ρ represents the value returned by the method.

Some examples of the usage of the `methodReturnValue` variable are given in Section 5.2.2.1.

5.2.1.2 Clean behaviour

Clean behaviour comprises the conjunction of a number of cleanness conditions, which, if they are all satisfied by all statements of the program, will guarantee that no exception conditions will occur. Thus, this property specifies the conditions that should be satisfied in order to ensure that abnormal conditions will never occur. A cleanness condition α_{exc} is defined as

$\alpha_{exc} \equiv$ <$\forall$ statements s **where** s in G $\wedge$ s contains $MessageExpression_{exc}$ ::
$Condition_{exc}$
>

where *exc* is the exception condition, $MessageExpression_{exc}$ is the message expression that contains the construct that could result in the exception condition and $Condition_{exc}$ is the condition that needs to hold in order to prevent the exception condition from occurring.

[1] In the case where the SLOOP program G shares variables with other programs running concurrently with program G, then if program G has reached a fixed point, the state of program G can be changed only by statements outside program G.

Clean behaviour is therefore represented by

invariant $\bigwedge_{exc} \alpha_{exc}$

The conjunction is taken over all possible categories of exception conditions *exc* in the program.

For example:

$\alpha_1 \equiv$ <$\forall$ statements *s* **where** *s* in $G \wedge s$ contains *exp1*/*exp2* :: $exp2 \neq 0$
>

where *exp1* and *exp2* represent two different expressions.

"The exception condition is division by zero."

$\alpha_2 \equiv$ <$\forall$ statements *s* **where** *s* in $G \wedge$ s contains *receiver* at: *index* ::
$1 \leq index \wedge index \leq$ (*receiver* capacity)
> "The exception condition is an out of bounds subscript."

$\alpha_3 \equiv$ <$\forall$ statements *s* **where** *s* in $G \wedge$ s contains *receiver message* ::
receiver notNil $\wedge$ *receiver* respondsTo: (*message* selector)
> "The exception condition is a message sent to a non-existent receiver or one that does not support the message selector."

The above definition applies to the design phase of a system. However, clean behaviour at the analysis level can be viewed as the specification of those properties that will guarantee that certain abnormal conditions will never occur. This is explained further in Section 5.4.1.2, where examples of such properties are also given.

5.2.1.3 Global and local invariants

A global invariant is defined as:

$(\varphi(\bar{x}) \Rightarrow p) \wedge$ **stable** *p*

which can be written as

invariant *p*.

A global invariant can be defined for a class, in which case the invariant must hold before and after the execution of each parallel statement of the parallel methods of the class, as well as before and after the execution of each sequential method of the class. A global invariant can also be defined for a parallel method, in which case the invariant must hold before and after the execution of each parallel statement of that parallel method. A global invariant only becomes effective after instance creation has taken place, i.e. it has to hold after the execution of the last statement of an instance creation method, but it does not have to hold before that.

The concept of a local invariant is not used in a SLOOP program, since universal quantification is used in the definition of SLOOP safety properties. Invariants are not associated with specific locations in a SLOOP program.

5.2.1.4 Mutual exclusion

As described in Chapter 4, mutual exclusion is achieved in the SLOOP method by encapsulating all the statements that comprise a critical section into a single SLOOP parallel statement, since each SLOOP parallel statement is executed atomically. This is possible because of the expressive power of SLOOP parallel statements, as discussed in Chapter 4. For example, if the requirement is to remove the first two consecutive members from an ordered collection, then the steps to perform this action represent a critical section. It has to be ensured that no other object can remove or add members to the ordered collection between the first and second *removeFirst* messages.

As was shown in Chapter 4, Section 4.2.3, this is achieved as follows: the *if* clause of the SLOOP parallel statement contains the test whether there are at least two elements in the collection. If the condition evaluates to true, the corresponding *component-part* of that same statement then invokes a method which sends the *removeFirst* message to the collection twice in succession. Thus, all the actions are contained in a single parallel statement.

When the SLOOP software development method is used, **mutual exclusion** properties are therefore **not specified**. Instead, **safe liveness** properties are used to indicate which actions should be performed as a single atomic unit. Safe liveness properties are discussed in Section 5.2.3.1. At this stage it suffices to say that a safe liveness property can be defined as *p* **ensures** *q*, which means that if predicate *p* holds, then predicate *q* is achieved via a **single** parallel statement.

5.2.1.5 Deadlock freedom

The conditions for deadlock and how they apply to the SLOOP environment were discussed at length in Section 4.3.6.5 of the previous chapter. In summary, it is not considered necessary to define deadlock freedom correctness properties at the design level. This is because there is no concept of processes at the design level. There is therefore no concept of a process which can be blocked while waiting for a response from another process. At the design level one only needs to think in terms of the parallel statements of the SLOOP program. The semantics of the SLOOP parallel statements are defined such that two parallel statements that share objects may not execute simultaneously.

As described in Section 4.3.6.5, a SLOOP program could possibly contain circular conditional parallel statements. However, since such circular conditions could also be discovered via the correctness arguments of the progress properties of the program (as demonstrated in Section 4.3.6.5), the SLOOP method advocates the specification of the appropriate progress properties as a more pragmatic way to handle this problem.

For the reasons given above it is therefore not considered necessary to define deadlock freedom in terms of SLOOP parallel statements.

At the implementation level, the possibility of deadlock does exist. The deadlock prevention strategy used at this level was discussed in Section 4.3.6.5 of the previous chapter and more detail is provided in Chapter 8.

5.2.1.6 Generalised deadlock freedom

Since a SLOOP program does not contain waiting locations which contain looping instructions, this property does not apply to SLOOP programs.

5.2.1.7 Unless property

An **unless** property is defined as:
p **unless** *q*

Thus, if predicate *p* holds, then after the execution of parallel statement *s* (which could include the execution of sequential method *SM*), predicate *p* either continues to hold or predicate *q* holds.

5.2.2 Liveness properties

5.2.2.1 Total correctness

The total correctness property is particularly relevant for the specification of a terminating program using the SLOOP method. When referring to the parallel statements of the program, it is represented by the following statement:

$\varphi(\bar{x})$ **leads-to** FP $\wedge$ $\psi(\bar{x},\bar{y})$

Thus, a fixed point will be reached and when it is reached the results will be correct.

For sequential methods, total correctness is defined as follows:

$\varphi(\bar{x})$ **results-in** (methodReturnValue = ρ) $\wedge$ $\psi(\bar{x},\bar{y})$,

where ρ represents the value returned by the method.

The `methodReturnValue` variable was discussed in Section 5.2.1.1. Examples of its usage are now presented. In the first example it is used to represent the value returned by the method. The method below determines whether there is an idle service provider to which a service request can be assigned. The method returns true if it finds a service provider that can accept a new service request of the specified service category and false otherwise. (The candidate service providers are elements of the `spSubset` collection.) The total correctness property is as follows:

```
true results-in methodReturnValue =
      (spSubset detect:
      [:each | each canAcceptNextSR: serviceQCategory]
      ifNone: [nil] ) notNil
```

Although all sequential methods return values, the main purpose of some methods may be to modify some variables and there is no requirement to return a specific value. In that case the receiver is returned as a default return value. This is the second scenario in which the `methodReturnValue` variable can be used. The following total correctness property provides an example:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
      results-in
            methodReturnValue = self ∧
            serviceRequest = aServiceRequest ∧
            newEventRequired ∧
            categoryIndex = (x + 1) \\ nrOfCategoriesServed
      >
```

It is the total correctness property of the `processServiceRequest:` method of the ServiceProviderSimulator class. This method is executed when a new service request is assigned to a service provider simulator. Its purpose is to update the `serviceRequest`, `newEventRequired` and `categoryIndex` instance variables as indicated above. The return value is not used by the sender. The methods associated with the above properties are discussed further in the chapters to follow and full SLOOP specifications of these methods appear in Appendix B, Sections B.10 and B.13 respectively.

5.2.2.2 Intermittent assertions

An intermittent assertion specifies that if property *p* holds at some stage, then at some later stage property *q* will hold. Thus:

p **leads-to** *q*

The important issue here is that p is not required to hold until q holds. If that is a requirement, the **ensures** or **until** relations have to be used. They are discussed in the precedence property category.

5.2.2.3 Accessibility

This is represented by:
R_1 **until** C_1
where R_1 represents the **condition** that holds when concurrent object 1 wishes to enter its critical section and where C_1 represents the **condition** which holds when concurrent object 1 is inside its critical section. Thus, concurrent object 1 will eventually enter the critical section associated with C_1 if condition R_1 becomes true and R_1 continues to hold until object 1 enters the critical section.

Accessibility and mutual exclusion properties are complementary properties. The former specifies the liveness aspect of critical section handling, whereas the latter specifies the safety aspect. As explained in Section 5.2.1.4, mutual exclusion properties are not specified in SLOOP programs. Instead, safe liveness properties are used to indicate which actions should be executed atomically.

A safe liveness property can be defined as p **ensures** q[2]. It specifies that a **single** parallel statement establishes q, which takes care of the safety aspect of the critical section handling, since each parallel statement executes atomically. The **ensures** logical relation also guarantees progress, which takes care of the liveness aspect of the critical section handling. As a result it is not necessary to specify accessibility properties for SLOOP programs, provided safe liveness properties are specified to indicate which actions should be executed atomically.

5.2.2.4 Liveness (absence of individual starvation)

All parallel statements are executed infinitely often, therefore all concurrent objects will always progress from one statement to the next. However, since a statement may execute conditionally, it could happen that the condition is always false whenever the statement is scheduled. In that case it amounts to starvation, since the state changes specified by the statement can never take place.

A design heuristic which addresses this problem is to design the conditions in such a way that once they evaluate to true, they remain true until the associated statement has been executed. However, this is not always possible, therefore this is not a hard and fast rule, but merely a heuristic. In SLOOP programs the intermittent assertions (described in Section 5.2.2.2) and the safe liveness properties (described in Section 5.2.3.1) should ensure that each concurrent object makes the desired progress.

5.2.2.5 Responsiveness

This property states that if the predicate r_i representing some request i is true at some point, then eventually predicate g_i will become true, representing the granting of request i. Thus,
r_i **leads-to** g_i.

---

[2] If it is not important that a **single** parallel statement should establish q, then safe liveness is defined as p **until** q, as described in Section 5.2.3.1.

5.2.3 Precedence properties

5.2.3.1 Safe liveness

If property p has to hold at least until q holds and a single parallel statement establishes q, the **ensures** relation is used, i.e.
p **ensures** q.

If property p has to hold at least until q holds, but q is not established by a single parallel statement, the property is formulated as follows:
p **until** q,
where the **until** logical relation is defined as follows:
p **until** $q \equiv (p$ **unless** $q) \wedge (p$ **leads-to** $q)$.

5.2.3.2 Absence of unsolicited response

Absence of unsolicited response is formulated as follows:
p **precedes** q,
where the precedes logical relation is defined as follows:
p **precedes** $q \equiv \neg((\neg p)$ **until** $q)$.

Thus, if q ever becomes true, it will not become true until p becomes true first [MaPn81a].

5.2.3.3 Fair responsiveness

Fair responsiveness means that if the request represented by r_1 is received before the request represented by r_2, then the first request is granted before the second one, represented by g_1 and g_2 respectively. In addition, the individual responsiveness properties hold as well. Thus,
$((r_1$ **precedes** $r_2) \Rightarrow (g_1$ **precedes** $g_2))$
$\wedge$
$((r_1$ **leads-to** $g_1) \wedge (r_2$ **leads-to** $g_2))$.

5.2.4 SLOOP checklist of correctness properties

The list of correctness properties that are used in SLOOP specifications, is now presented, based on the discussions above. In a SLOOP specification each correctness property has a unique number associated with it. This number is used for cross-referencing purposes. The number has the following format: XYx-yy, where

X	the value A, D or I denotes that the property emanates from the analysis, design or implementation phase respectively,
Y	has the value S (safety), L (liveness) or P (precedence),
x	is the number of the property type within the safety, liveness or precedence category,
yy	distinguishes the property from others of the same type.

The X and yy generic symbols only receive values once an actual correctness property is specified for a system or class.

Safety properties:
XS1-yy: Partial correctness
XS2-yy: Clean behaviour
XS3-yy: Global invariants
XS4-yy: Unless property

Liveness properties:
XL1-yy: Total correctness
XL2-yy: Intermittent assertions
XL3-yy: Responsiveness

Precedence properties:
XP1-yy: Safe liveness
XP2-yy: Absence of unsolicited response
XP3-yy: Fair responsiveness

This checklist will be used in the example that is described in the remainder of this chapter.

5.3 Constructing the class diagram

In the previous chapter a number of aspects of the SLOOP method were illustrated via the call centre example. The requirements for a call centre system are now discussed in more detail. The call centre system serves as a case study of the application of the SLOOP method and is used in this and the remaining chapters to highlight various features of the method.

This specific example was chosen because it is non-trivial and it therefore demonstrates how the SLOOP method addresses the **scalability** problem. At a high level of abstraction the functionality is not restricted to a call centre; any system which has to switch service requests to service providers based on certain criteria can be based on the framework of high level classes presented here. This **reusability** aspect is discussed in more detail in Chapter 9.

For the sake of brevity, the example is restricted to the first level of abstraction.

5.3.1 Initial informal problem statement

The problem statement below refers to a specific type of call centre, viz. one with anonymous service users. A typical application of such a call centre is the toll-free customer service offered by many large companies. The identity of the caller is not required in order to **switch**[3] the service request to the appropriate service provider. For example, a potential customer might just want to enquire where the company branch in a certain area is located. Another customer might want to order an item, in which case information about the customer is required for payment purposes, but this information is obtained by the service provider, not the call centre, since the latter does not require it for switching purposes.

There are other types of call centres that use the Automatic Number Identification (ANI) service provided by the Public Switched Telephone Network (PSTN) to identify the caller, which influences the way in which the call centre switches the service request. For example, if the caller is identified as an important customer, the service request might be enqueued in a high priority queue. However, since the **purpose** of this case study is to illustrate various aspects of the SLOOP method and not to provide a comprehensive framework for call centre systems, the problem statement below refers to anonymous service users only. In the next chapter (Section 6.7) it is illustrated how the approach that is followed during the design phase allows for relatively simple extensions to cater for other types of call centres. It therefore demonstrates how the goals of **reusability** and **extensibility** are addressed.

[3] The term 'switch' is used to refer to the actions performed by the call centre when it analyses the service request in order to determine which service provider (or set of service providers) would be most suitable to service the request, as well as all the subsequent actions taken by the call centre that culminate in a connection between the service user and a service provider. The service request is an object created by the call centre and it contains all the information required by the call centre about a particular request for service.

The initial informal problem statement is as follows:

A call centre is a non-terminating system which ensures that dial-in users are serviced in a specified order by the available service providers. This is accomplished as follows:

The service user places a call to a central (usually toll-free) number. Physically, this number may represent several lines (if line hunting is provided by the Public Switched Telephone Network), or it may represent a single high speed line which can carry multiple connections. The number of calls that can be handled **simultaneously** is therefore bounded. If the service user receives a ringing tone, it implies that a connection or line is available, otherwise a busy tone is signalled. The call centre is not responsible for controlling the ringing and busy tones. That is performed by the Public Switched Telephone Network (PSTN).

The line or connection that is occupied by a service user remains so for the duration of the call, i.e. until the user hangs up, the connection is terminated due to a problem or the service provider has finished serving the user. The identity of the service user is irrelevant, therefore there is no upper limit to the number of service users, but they cannot all connect to the call centre simultaneously. Since the call centre does not store any information about the service users for switching purposes, there are no physical constraints regarding the maximum number of service users that can be supported.

The call centre processes the calls on a first-come, first-served basis, i.e. as they are received from the PBX. If the call centre supports more than one service request category, it receives information about the type of service required by the service user together with the call. This information is obtained from the service user via an Interactive Voice Response (IVR) before the call reaches the call centre, as shown in Figure 5-1. The call centre uses this information to categorise the service required by the service user. If at least one service provider is active which provides a service of the specified category, the service request is accepted, otherwise it is rejected and the connection is terminated.

The call centre ensures that each service user is serviced on a first-come, first-served basis within each service request category. This is illustrated graphically in Figure 5-1. It shows that there are a number of queues (one for each service request category). Each service request is entered into one of these queues and each queue is serviced on a first-come, first-served basis.

Once the service request has been allocated to a service queue, the service user is informed at regular intervals of his/her progress towards being served. If there is only one service request category, there is no need to obtain information from the service user in order to categorise the service required.

The order in which the various service request categories are served is based on some application-dependent criteria. For example, there may be high and low priority categories and the criterium might be to serve the high priority category before the low priority category as long as there are high priority service requests pending. Alternatively, the criteria might merely bias service towards the high priority category in some way.

There are one or more service providers. The number of service providers is bounded. Each service provider may service one or more service request categories. For example, one type of service provider (say the elementary type) might service queries only. Another type of service provider (say the advanced type) might be able to service both queries and other transactions. There are therefore one or more service provider categories. Each service request category should be serviced by at least one service provider category.

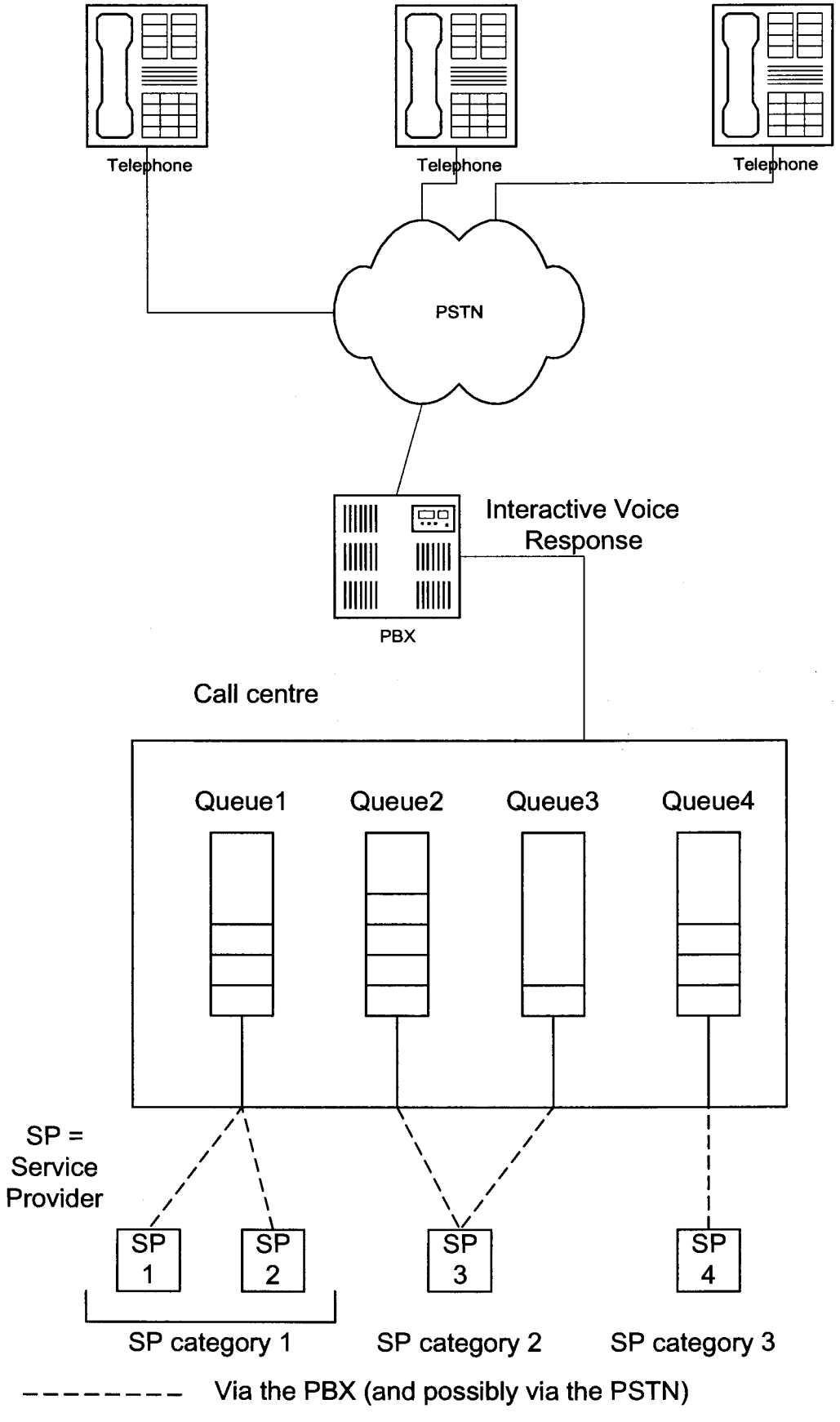

Figure 5-1. Architecture of a call centre and its environment at a high level of abstraction.

Figure 5-1 illustrates some of the allowed combinations as follows: Both Service Provider 1 (SP1) and SP2 service Queue 1. Queue 1 represents local directory queries. Queue 1 is serviced by service providers in the elementary category. Both SP1 and SP2 belong to the elementary service provider category. Queues 2 and 3 represent non-local national directory queries and international directory queries respectively. Both queues are serviced by service providers in the intermediate category. Only SP3 belongs to this category. Queue 4 represents other transactions and is serviced by service providers in the advanced category. SP4 belongs to that category.

The service request categories and the service provider categories referred to in this problem statement are devised in order to provide flexibility in the way in which service requests are allocated to service providers. These categories may therefore be independent of any other service request categorisation performed by the service providers once a service request is allocated to one of them.

Once a service request has been categorised and enqueued in a service queue, the call centre ensures that it is eventually assigned to a service provider, unless the service user hangs up beforehand. The nature of the service being provided is irrelevant to the call centre. The connections between the service providers and the call centre may be via the PSTN or they may be established internally via the local PBX as shown in Figure 5-1.

The initial product needs to be tested by simulating the actions of the service users, service providers and the communication provider.

System boundaries:
The service providers, service users and communication service form part of the environment of the call centre, i.e. the latter performs a queuing and switching function only. The service providers contain the software to process a service request. This software may vary from application to application and therefore does not form part of the call centre. The service providers may even be supplied by a third party. The software that needs to be designed is restricted to the part pertaining to the call centre and its interface with its environment. This is the part that remains unchanged between applications.

Assumptions:
At this level of abstraction a reliable communication medium is assumed between the service users and the call centre as well as between the call centre and the service providers.

The analysis phase is an **iterative** process as was indicated in Figure 4-9 in the previous chapter. While listing the properties of the system, it quite often happens that **deficiencies** in the problem statement are discovered. This is illustrated as the analysis for the above example is performed.

5.3.2 The class diagram

The first step towards creating a class diagram is the identification of the classes in the problem domain. The SLOOP method does not prescribe a specific technique towards accomplishing this, since the distinguishing aspect of the SLOOP method is its description of the **dynamic** behaviour of the system. Any of the usual techniques will suffice. For example, the nouns in the problem description can be extracted and used as a basis for identifying classes [RBPEL91]. This aspect of the analysis phase is not pursued further; only the results are given here.

A preliminary analysis of the problem statement yields the classes as shown in Table 5-1. It contains all the classes in the problem domain, i.e. those that form part of the system, (the call centre) as well as those in its environment. The simulation classes are included in addition to their real-world counterparts.

Class	Justification
ServiceUser	This class represents the service user. The service users are external to the call centre, i.e. form part of the environment of the call centre. There is no container class for the service users, which **models** the fact that there is **no upper limit** to the number of service users.
Connection	Although the problem statement refers to lines that are connected to the call centre and to calls that are placed, the crux of the matter is the fact that some or other mechanism is used to **connect** to the call centre. It is sufficient to model this at a higher level of abstraction via the Connection class. This ensures that the system is **not over-specified,** i.e. it leaves scope for any type of connection to the system. Once the connection is switched to a service provider, it represents the complete path from the service user to the service provider.
Connection Container	The aspect that needs to be modelled here is the fact that the number of **simultaneous** service user connections is **bounded**. ConnectionContainer, a container class with a maximum capacity, adequately describes this requirement. The ringing and busy tone objects do not yield new classes, because they refer to implementation detail of the communication service. At this level of **abstraction** it is only relevant to know whether a connection can be established or not, which is modelled by the states of the connections in the connection container.
Communication Provider	The CommunicationProvider class provides the communication service. It forms part of the environment of the call centre and could be the PSTN or in the case of service provider access, it could be private lines or the PSTN.
ServiceProvider	A service provider ultimately services the service request. The service providers form part of the environment of the call centre.
ServiceProvider Container	The ServiceProviderContainer class, which has a maximum capacity, **models** the physical **constraint** on the number of ServiceProvider instances that can be supported by the call centre. The very fact that service requests are queued implies the **boundedness** of the number of service providers. If there had been an unlimited number of service providers, each service request could have been assigned to a service provider as it was categorised. This is therefore an **analysis level** constraint and not a design level one.
ServiceCategory Allocator	The problem statement refers to the call centre when it describes the required behaviour of the system. Various aspects of the behaviour are modelled by various different classes. The ServiceCategoryAllocator class models that part of the call centre which allocates new service requests to the service queues of the appropriate service categories.
InputQueue	In this case there is no real-world object that needs to be modelled; instead the **abstract concept** of the **order** in which new service requests are processed needs to be reflected. This order is modelled by the properties of InputQueue, a container class which provides methods to manipulate its elements in a **First In First Out** manner. An alternative model which comes to mind is to use the ConnectionContainer class to represent this ordering. Thus, a Connection instance is added to the connection container in the order in which the connections are established. However, this also implies that the Connection instance has to be removed from the container as soon as the connection has been enqueued in a service queue, even though the

	connection is still in use. This means that the connection container no longer fulfils its other purpose, namely that of representing the boundedness of the number of users that may be connected to the system simultaneously. (The reason why the Connection instance has to be removed from the connection container once it has been enqueued in a service queue if there is no InputQueue class is to guarantee that the connection container class models the First In First Out ordering. Consider the following counter-example: If the Connection instance had remained in the connection container until the connection had been terminated, then the state of the Connection instance could have been used to indicate whether the associated service request had already been enqueued. However, since connections can be terminated in any order, new connections would have had to be entered into the first available slot in the connection container, in which case the elements of the connection container might no longer have been ordered in a First In First Out order. For example, slot 4 might have become available before slot 1.)
ServiceRequest	The ServiceRequest class represents all the information that the call centre requires from the service user. The ServiceRequest class is therefore an abstraction for identification information (in other call centre types), the type of service that is required and any other information required from the service user. The mechanism whereby this information is obtained is beyond the scope of the call centre (e.g. some of it may be supplied by the PSTN, other information may be provided via the IVR). A service request is associated with a connection.
ServiceCategory	This is an **aggregate** class. There is a ServiceCategory instance for each service request category. The number of service categories is also **bounded**, but that is a **design level constraint**. Nothing in the problem statement implies that there is an upper limit to the number of service categories. **At the analysis level there is therefore no ServiceCategoryContainer class.** The ServiceCategory has the following components: ServiceQueue, ServiceProviderSubset andServiceProviderCategories. The ServiceQueue instance contains service requests that are of the same category as that of the ServiceCategory instance. The ServiceProviderSubset instance contains references to the service providers that handle service requests of that particular category. The ServiceProviderCategories instance contains a set of service provider categories. These are the allowed categories of the service providers that may service the requests in the service queue belonging to this ServiceCategory instance. A ServiceCategory instance therefore has a service request category, an ordered collection of service requests, a non-empty collection of service providers and a non-empty collection of service provider categories associated with it. The service providers associated with a ServiceCategory instance do not all have to be of the same service provider category. The ServiceCategory instance monitors the status of the service providers referenced by the elements of the ServiceProviderContainer.

	The order in which the various service queues are serviced by the service providers is based on application-dependent criteria. Each ServiceCategory instance is responsible for determining whether the service requests in its service queue can be serviced and which service provider the service request should be allocated to.
ServiceQueue	The ServiceQueue class is a **component** of the ServiceCategory class. The service queue contains an ordered collection of service requests of the specified category. The elements of the collection are processed on a first-come first-served basis. Query and transaction are examples of service request categories.
ServiceProvider Subset	The ServiceProviderSubset class is a **component** of the ServiceCategory class. It contains a collection of references to all the service providers that handle service requests of the specified category.
ServiceProvider Categories	The ServiceProviderCategories class is a **component** of the ServiceCategory class. It contains a collection of all the service provider categories associated with the service request category.
TimerServices	When a service request is added to a service queue, a progress timer is started. When it expires, the service user is informed of the progress of the service request and the timer is restarted. Timers are also used by the simulation classes. The ServiceProviderSimulator uses timers to simulate the non-instantaneous nature of actions such as the servicing of a request. The CommsProviderSimulator uses timers to simulate the random periods that may elapse between receiving new calls. The TimerServices class handles all timer-related aspects.
CommsProvider Simulator	The system facilitates testing without actual service users, service providers and a communication provider by introducing classes to simulate their behaviour. The CommsProviderSimulator class **simulates** the actions of the communication provider.
ServiceProvider Simulator	The ServiceProviderSimulator class **simulates** the actions of a service provider.
ServiceProvider Simulator Container	The ServiceProviderSimulatorContainer class, which has a maximum capacity, models the physical constraint on the number of ServiceProviderSimulator instances that can be supported by the call centre.

Table 5-1. Classes yielded during the analysis phase.

Note: There is no need to create a ServiceUserSimulator class to simulate the behaviour of the service user. The reasons are as follows: Before the call is switched, the interaction with the service user occurs between the service user and the communication provider (e.g. via the IVR). After the call is switched, the interaction is between the service user and the service provider. There is therefore no direct interaction between the service user and the call centre at any stage.

Figure 5-2 shows the class diagram (static structure) of the problem domain. It does not contain the simulation classes. Figure 5-3 shows the class diagram where the simulation classes replace their real-world counterparts. The differences between the two diagrams are shown in bold.

The following is a very brief summary of the UML notation used in the diagrams. Classes are represented by rectangles. A solid line connecting two classes represents the association between the two classes. The small black solid triangle is the UML symbol which indicates in which direction to read an association name. The end where the association connects to a class is called an association role. An association role is adorned with a multiplicity string. The latter specifies the allowable cardinalities of a set. An unlimited upper bound is denoted

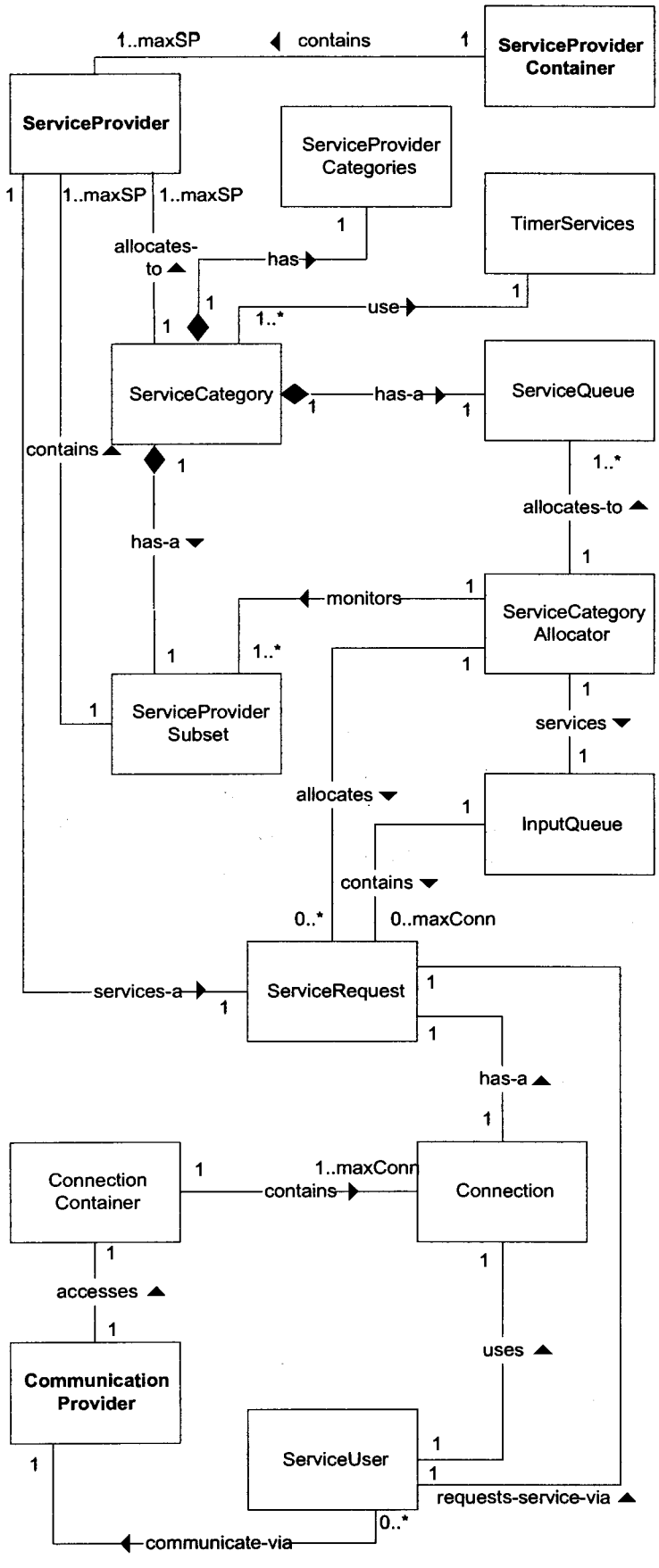

Figure 5-2. Class diagram of the call centre and its environment (without simulation classes).

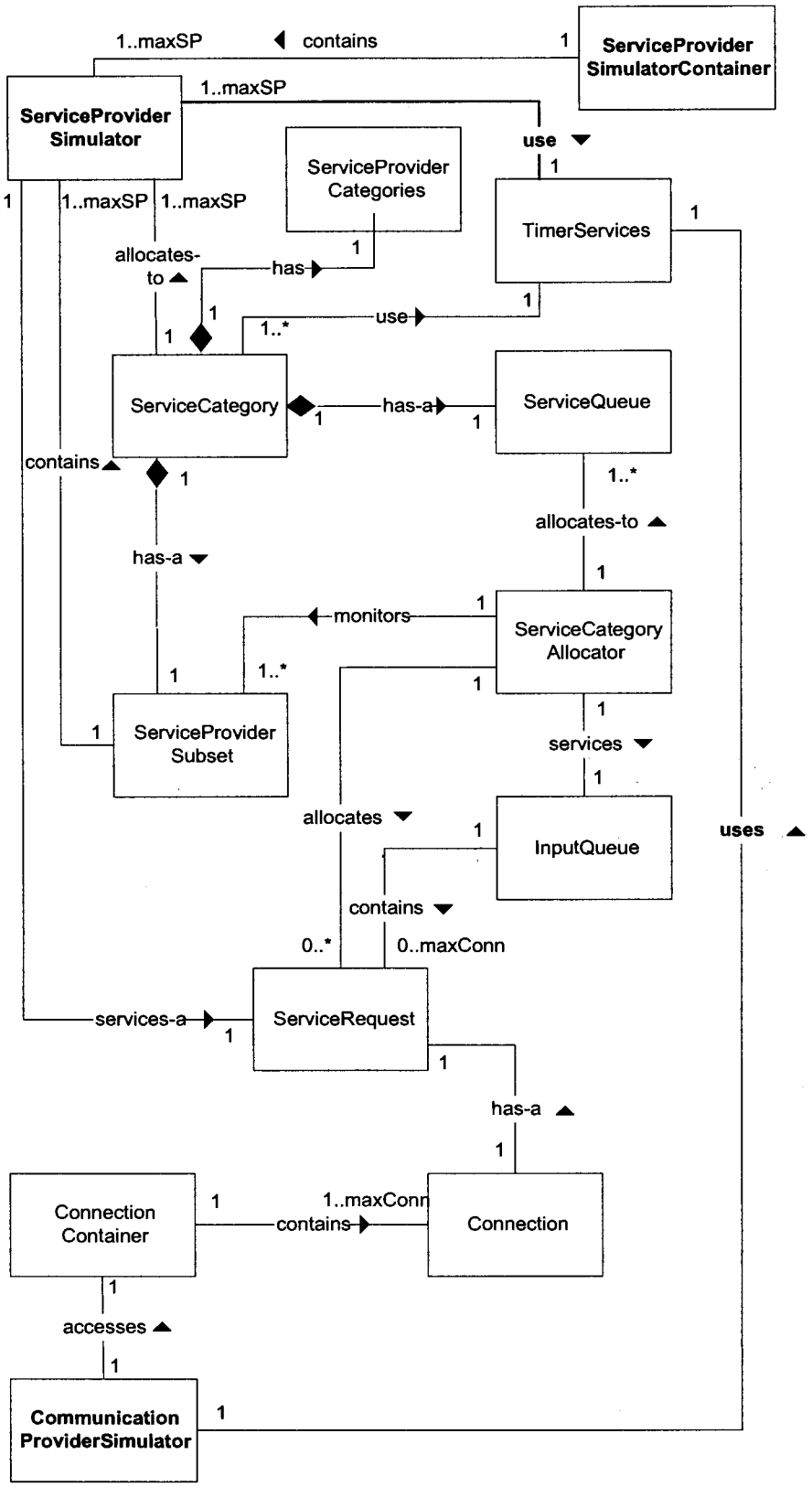

Figure 5-3. Class diagram of the call centre and its environment (with simulation classes).

by specifying an asterisk (*) for the upper bound value. If an association role is adorned with a diamond, it indicates aggregation. The diamond is attached to the aggregate class. If the diamond is filled, it indicates composition, the strong form of aggregation.

The classes in the class diagram represent **real-world objects** as well as **abstract concepts** that are evident from the analysis of the problem statement. The suitability of these classes will be re-inforced during the specification of the system properties. If necessary, the class diagram is modified if it is found that it is inadequate at that stage. This iterative process was reflected in Figure 4-9, which showed a graphical representation of the requirements analysis phase.

As stated above, the purpose of this chapter is to illustrate how the **behaviour** of the system is specified via correctness properties during the analysis phase. The class diagram is therefore not elaborated any further. For example, a data dictionary is not provided here and the attributes of the various classes are not described. Attributes that are used in the specification of the correctness properties below will be explained where necessary.

5.4 Specifying system behaviour informally

The next step is to list the desired properties of the system informally. These properties result from an inspection of the informal problem statement using a checklist of correctness properties such as the one given in Section 5.2.4. Initially the focus is on global system properties as opposed to individual class properties. These include, inter alia, global invariants as well as properties that describe the nature of the **interactions between** the objects. This step yields some of the methods of the classes defined earlier.

By listing the system properties it can be determined what is required of the constituent classes. Subsequently, the properties displayed by each individual class are defined. Note that the SLOOP syntax does not have a special construct for the specification of system properties. This is because the system itself can also be viewed as a class and the properties can therefore be included in the class properties section. This approach is exemplified in Chapter 6.

5.4.1 Safety properties

The four safety (invariance) properties listed in Section 5.2.4 are now considered in turn. The properties that are identified here emanate from the requirements analysis phase, therefore all the property identifiers start with the letter 'A', followed by the letter 'S' to indicate that they are safety properties. Four groups are considered in turn, viz. partial correctness (AS1-yy), clean behaviour (AS2-yy), global invariants (AS3-yy) and unless properties (AS4-yy).

5.4.1.1 AS1-yy (partial correctness)

The system is cyclic (does not terminate), therefore partial correctness does not apply.

5.4.1.2 AS2-yy (clean behaviour)

At the **design level**, the clean behaviour properties ensure that **no exceptions** will be raised as described in Section 5.2.1.2. At the **analysis level** clean behaviour properties specify the conditions that should always be satisfied in order to **prevent violations of the specified system limits from occurring**. For example, the informal problem statement specifies that the number of simultaneous connections supported by the system is bounded. The call centre system should therefore not accept a new connection if the maximum number of connections are already established. This requirement results in the specification of the following clean behaviour property:

AS2-01. The connection container has a maximum capacity of maxConn connections, where maxConn is a positive integer.

Property *AS2-01* confirms the choice of a ConnectionContainer class in the class diagram; it is used to model the **capacity restrictions** of the call centre. The number of elements in a connection container (its size) may never exceed its capacity. The communication provider has to ensure that this property holds when it allows a connection to be established, otherwise the connection has to be refused. The system has to support at least one connection in order to provide any service at all, therefore `maxConn` has to be a positive integer.

The phrase "where maxConn is a positive integer" naturally leads to the following observation: The informal problem statement seems to assume that at least one connection will be supported, but there is no explicit reference to the **minimum requirements** of the system. This **reveals a deficiency** in the informal problem statement. It also leads to another question: What other collection classes in the class diagram are affected by the specification of a minimum capacity? This also reveals that the informal problem statement does not address the issue of the behaviour of the call centre system in the case of **resource problems**.

As a result of the above analysis of the problem statement, consultation with the client is required. An improved informal problem statement is produced. The following requirements are added:

"The call centre has a required capacity of maxConn connections, where maxConn is a positive integer. Once a connection has been accepted by the call centre, the associated service request may only be rejected if none of the active service providers have the capability to service the request. The service request may not be rejected due to any other resource problems.

The call centre has to support maxCategories of service request categories. It has to support maxSP of service providers."

As a result of the above modification to the original problem statement, the class diagram is modified to include a ServiceCategoryContainer class. Thus, now there is an analysis level requirement to model the fact that a certain number of ServiceCategory instances have to be supported. Property *AS2-01* is modified as follows:

*AS2-01. The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer.*

The following additional clean behaviour properties are specified:

AS2-02. The minimum capacity of the input queue is equal to the capacity of the connection container.

The InputQueue class models the order in which the service requests are received. It is therefore essential to be able to represent a service request in the input queue if it is possible to represent its associated connection in the connection container. Note that the property specifies that the capacity has to be greater than or equal to that of the connection container (a **minimum** capacity is required). This ensures that the property is not overspecified. There is no requirement for the input queue to be larger than the minimum, but there is also no analysis level reason why it should be restricted to have exactly the same capacity as the connection container. This enables one to consider multiple design options, as will be seen in Chapter 6, Section 6.2.1.

AS2-03. The minimum capacity of each service queue is equal to the capacity of the connection container.

The above property specifies that there will be space for a service request in a service queue if its associated connection is represented in the connection container. Thus, even if the previous `maxConn-1` service requests are of the **same** category and have not been allocated to a service provider yet, the capacity of the service queue will be sufficient to hold another service request.

This property ensures that the processing of a connection and the placement of its associated service request into a service queue are **independent** of any other service requests in the system. The following scenario therefore **cannot** occur: Suppose there is a high priority and a low priority service request category. The maximum capacity of each queue is `maxConn/2`. The low priority queue contains `maxConn/2` service requests and the high priority queue contains none. Service request A, which is of the low priority, arrives and is entered into the input queue. Since the low priority service queue is full, service request A has to remain in the input queue until space becomes available in the low priority service queue. If, however, one of the `maxConn/2` previous service requests had been a high priority service request, this service request would have been enqueued immediately.

The above scenario also illustrates that a service request can be delayed by a service request from a different category if property *AS2-03* is **not** specified. If, for example, service request A in the above scenario is followed by a high priority service request B in the input queue, then the latter cannot be processed until space becomes available in the low priority service queue and service request A can be removed from the input queue.

If property *AS2-03* is satisfied, then the above scenarios are not possible. Note that this is **not** a **design level** correctness property. It **does not prescribe how** the service queues should be implemented; it merely specifies that each service queue should be allowed to reach the size of `maxConn` in order to ensure that the service requests in the input queue can be serviced in a First In First Out order while at the same time ensuring that one service request cannot delay another while they are both in the input queue. This is reflected via the following additional sentence in the informal problem statement:
"The call centre ensures that service requests of one category are not affected by inadequate resources allocated to service requests of another category."

In order to reason about the minimum capacities of the input queue and the service queues, properties *AS2-04* and *AS2-05* as given below are also required. These properties ensure that obsolete entries are not allowed in these queues. If these properties are not satisfied, then it cannot be guaranteed that the system will be able to process a service request associated with a connection that is represented in the connection container.

AS2-04. If a connection is terminated, it implies that its associated service request is not present in the input queue.

AS2-05. If a connection is terminated, it implies that no service queue contains a service request associated with the connection.

The property below ensures that no obsolete service requests are associated with a service provider.

AS2-06. If a connection is terminated, it implies that no service provider is servicing a service request associated with the connection.

The next two properties demonstrate how the need to formulate correctness properties results in the clarification of ambiguous statements in the informal problem statement. Consider the following two sentences that were added to the informal problem statement earlier in this section: **"The call centre has to support maxCategories of service request categories. It has to support maxSP of service providers."**

It is not clear from the above whether the word 'support' should be interpreted that the relevant container objects should merely **be able** to handle the specified maximum number of elements, or whether those elements should **always be present** after initialisation. It is important for the person formulating the correctness properties to know exactly how the word 'support' should be interpreted, since it determines whether the word 'size' or 'capacity' should be used in the correctness properties. (The **size** of a collection reflects the **number of elements** of a collection, whereas the **capacity** indicates the **largest** number of elements that the collection **may contain**.)

Again the informal problem statement needs to be updated to clarify this issue. The following sentence is therefore added:
"All supported service categories and service providers must exist after initialisation of the call centre."

Properties *AS2-07* and *AS2-08* can now be formulated to reflect this interpretation:

AS2-07. *The size of the service category container is equal to maxCategories, where maxCategories is a positive integer.*

AS2-08. *The size of the service provider container is equal to maxSP, where maxSP is a positive integer.*

Since the above two properties refer to the 'size' rather than the 'capacity' of the respective containers, they make it clear that the elements of the containers should exist once the system has been initialised. The properties related to the maximum number of connections do not require the existence of instances of the Connection or ServiceRequest classes. They merely require that the classes that may **contain** instances of the Connection or ServiceRequest classes have sufficient **capacity** to **hold** these instances, hence the use of the word 'capacity' in those properties.

Property *AS2-08* implies that the number of service providers supported is bounded. This exposes another shortcoming of the informal problem statement: although the problem statement refers to service provider simulators, it does not specify the number of service provider simulators that should be supported.

Another sentence therefore needs to be added to the informal problem statement to clarify the role of the service provider simulators:
"In the case where the service providers are simulated, the service providers are replaced by maxSP of service provider simulators."

As a result, property *AS2-09* is formulated:

AS2-09. *The service provider container contains either service providers or service provider simulators.*

Note: In the remainder of this chapter, the terms 'service providers' and 'service provider simulators' are used interchangeably, unless otherwise stated.

The last clean behaviour property is the following:

AS2-10. The service provider subset of each service category contains at least one (simulated) service provider instance.

Property *AS2-10* is an example of an invariant that can only be satisfied after system initialisation has been completed. If the ServiceProviderSubset instance has been created, but none of the service provider instances have been created yet, then this property does not yet hold. Due to dependencies such as this one, invariants are not required to hold until after system initialisation has been completed.

Property *AS2-10* does not require the service providers in the service provider subset to be **active**. If the service provider subset contains at least one element, it implies that at least one service provider exists which has the **capability** to process service requests of that service category. The status, i.e. whether it is active or not, is an attribute of the service provider and it indicates whether the latter is presently able to process service requests. Property *AP1-04* deals with the impact of the status of the service provider on the handling of a service request.

5.4.1.3 AS3-yy (global invariants)

Global invariants describe the properties that should always be satisfied in order to ensure that something bad will never happen. In many cases **undesired behaviour** is **not described explicitly** in an **informal problem statement**, since the focus there is to describe the desired functionality. However, in order to ensure that undesired behaviour is **prevented**, one has to consider what such behaviour could possibly be. This demonstrates one of the advantages of a software development approach which focuses on correctness properties: it forces one to consider the situations that should be prevented as well as those that should occur. Such a **"constructive" approach** results in a product that is more reliable from the outset, since many of the problems that are often only discovered during the testing phase are now considered during the earlier phases of development.

The global invariants presented next are identified based on the informal problem statement given in Section 5.3.1. In some cases the properties can be inferred directly from the problem statement, but in other cases the requirement is implicit, as will be pointed out where relevant.

AS3-01. The establishment of a new connection implies that the associated service request is appended to the input queue.

The InputQueue class is used to model the First In First Out order in which service requests are handled by the call centre. The above property is required in order to ensure that the order of the elements of the input queue does indeed reflect the order in which the connections are established.

AS3-02. The category of each service request in a service queue is the same as the service category associated with the service queue.

AS3-03. Each service category is unique.

Properties *AS3-02* and *AS3-03* describe the correctness properties that are required in order to ensure that a service request is always placed into the correct service queue. Property *AS3-04*, presented next, can be derived from the above two properties.

AS3-04. A service request is only present in a single service queue at a time.

Property *AS3-04* is one of the properties that are required in order to ensure that a service request is not serviced in **duplicate**. The other properties are *AS3-05* and *AS3-06*. Note that the possibility of duplicate service requests is not mentioned at all in the informal problem statement.

It is assumed that the system will behave correctly and that includes not generating duplicate service requests. The **advantage** of formulating correctness properties is the fact that these **assumptions** are now being made **explicit**.

AS3-05.A service request is only present in the input queue if it has not been allocated to a service queue.

Thus, once it is allocated to a service queue, it is removed from the input queue. This property also ensures that service requests are not serviced in duplicate, but at a different level. It guarantees that the service request will not be entered into a service queue more than once.

AS3-06. A service request is only present in a service queue if it has not been serviced, i.e. if it has not been assigned to an element of the service provider subset.

Thus, once it is allocated to a service provider, it is removed from the service queue. This property also ensures that service requests are not serviced in duplicate. Should a service request not be removed, then it could first be assigned to SP1 and then to SP2. Both service providers could then proceed to service the request concurrently, which is not the desired behaviour. The following sentence is added to the informal problem statement:

AS3-07. Adding a service request to a service queue implies that a progress timer is started for the service request.

AS3-08. The category of each element of the service provider subset has to match one of the elements of the ServiceProviderCategories instance associated with the service category.

Phrased differently, property *AS3-08* specifies that the category of each service provider that may service requests associated with a particular service category, must match one of the service provider categories contained in the ServiceProviderCategories instance associated with that particular service category.

The combination of properties *AS3-03* and *AS3-08* indicates that a service provider category does not necessarily correspond with a service request category. This implies that the service providers can be designed with a categorisation that is independent from the categorisation of the service requests. The call centre is responsible for mapping the two types of categorisations. Associating a collection of service provider categories with each service category facilitates this and property *AS3-08* ensures that the mapping is performed correctly.

AS3-09. A service provider / service provider simulator services a single service request at a time.

Whereas properties *AS3-04* to *AS3-06* ensure that service requests are not serviced in duplicate, this property ensures that a service request **does not get lost**. Thus, once a service request is allocated to an element of a service provider subset, no other service request can be allocated to that element until the latter has completed servicing the first service request. It is therefore not possible to overwrite the first service request. This is another aspect of the call centre behaviour that is not described explicitly in the informal problem statement. The following sentence is therefore added to it:
"The call centre ensures that no service request is duplicated or lost."

The next invariant specifies that a service request may only be assigned to a service provider that can handle requests of the service category to which the service request belongs.

AS3-10. A service request may only be assigned to an element of the service provider subset associated with the service request category that matches the category of the service request.

The last invariant deals with the fact that a service provider can service multiple service categories.

AS3-11. A service provider / service provider simulator may be an element of multiple service provider subsets simultaneously.

The above global invariants were derived by considering every sentence of the informal problem statement and deciding whether it implied invariant behaviour. This included specifying desirable behaviour as well as specifying properties to prevent undesirable behaviour. It cannot be claimed that the list is complete (this is difficult when using a logic-based method [JiZh96] as was discussed in Section 2.4.3 of Chapter 2), but at least it provides a basis for reasoning about the correctness of the system in terms of those properties that have been specified. It also encourages the software designer to consider more than just those aspects that are explicitly mentioned in the problem statement, as was demonstrated by several of the properties listed above.

5.4.1.4 AS4-yy (unless properties)

The issue of the availability of the service providers is raised when one considers the implications of the following sentence in the problem statement: "Once a service request has been categorised and enqueued in a service queue, the call centre ensures that it is eventually assigned to a service provider, unless the service user hangs up beforehand." Thus, once a service request has been accepted by the call centre and enqueued in a service queue, it is guaranteed to be serviced by a service provider. The availability of a service provider therefore has to be guaranteed from that point onwards. Again, the initial problem statement is inadequate, since it does not specify explicitly whether service providers may go out of service and under what conditions. It is therefore updated as follows:

"It is assumed that if a service provider is available at the time when a service request is allocated to a service queue, but subsequently needs to go out of service, it remains available in a restricted fashion until that service request has been processed. The restriction is that it will not process any service requests that were added to the service queue after the service provider had made its intention to go out of service known. A service provider that operates in such a restricted mode may go out of service before the specified service request has been serviced only if at least one other service provider that can provide the specified service is available."

Careful analysis of this requirement yields a third service provider state, viz. restricted-active. This state is introduced to enable a service provider to indicate its intention of going out of service, while still honouring its commitments regarding service requests **already enqueued** in the relevant service queues. Thus, a service provider can either be active, non-active or active but operating in a restricted mode.

This requirement results in the following unless property:

AS4-01. If the number of active or restricted-active elements of a service provider subset is greater than zero, then it remains greater than zero unless the service queue associated with that service category is empty.

Thus, if at least one element of a service provider subset is active or restricted-active, then a minimum of one element must remain active or restricted-active for at least as long as there are elements in the service queue associated with that service category.

Note that it is not specified that an active / restricted-active element of the service provider subset has to remain active / restricted-active unless the service queue associated with the service category is empty. Such a property would be too restrictive. The way the property is formulated now, leaves room for many implementations. For example, one option is to implement the service provider in such a way that it may only go out of service if the service queues that it is servicing are empty. Another possibility is to allow a service provider to go out of service even if the affected service queues are non-empty, provided another service provider is active and able to service those requests. Yet another option is to allow the service provider to go out of service only if all the service requests that had been enqueued in the relevant service queues prior to its changing to the restricted-active state, have been serviced.

The unless property discussed in this section illustrates the need to consider the impact of **overspecification**. Since the analysis level properties should still hold once the design and implementation phases are reached, the analysis level properties should be sufficiently generic in order to allow for multiple design and implementation options.

5.4.2 Liveness properties

The three liveness (eventuality) properties listed in Section 5.2.4 are now considered, viz.
AL1-yy (total correctness),
AL2-yy (intermittent assertions) and
AL3-yy (responsiveness).

It is important that the software designer should keep the **SLOOP computational model** in mind throughout the specification of correctness properties, but the impact of this model is particularly evident in the **style** of the liveness and precedence properties. There is **no reference to flow of control**. The properties do not apply to specific program locations; once a SLOOP program is derived, they are quantified over all the statements in that program.

5.4.2.1 AL1-yy (total correctness)

The call centre software is cyclic (does not terminate), therefore total correctness does not apply to the system. Total correctness properties are defined for the sequential methods of the individual classes as shown in Section 5.4.5.1.

5.4.2.2 AL2-yy (intermittent assertions)

Intermittent assertion properties are specifically applicable to cyclic systems. It is used to specify that predicate p will eventually be followed by predicate q. In order to identify the relevant liveness properties, the informal problem statement is analysed in terms of its dynamic requirements. The general approach is to consider each class appearing in the class diagram and to scrutinise the problem statement for descriptions of progress related to that class. As will be evident from Section 5.4.3.1, most of the liveness properties in the call centre example are classified as safe liveness properties. In the case of a safe liveness property it is important that the predicate p should hold until predicate q holds.

AL2-01. Once a timer has been started, it will eventually expire or it will be stopped.

AL2-02. When a simulation event is required, a simulation timer is eventually started .

AL2-03. When a simulation timer expires, the simulation object eventually generates an event .

5.4.2.3 AL3-yy (responsiveness)

AL3-01. Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of the service provider container, or the service category allocator will reject the service request or the service user will hang up beforehand.

This property reveals a shortcoming in the informal problem statement. The latter specifies that an application may base the order in which the various service categories are served on some application-dependent criteria. It gives an example of a system which may support high and low priority categories and which might have a criterium to serve the high priority category before the low priority category as long as there are high priority service requests pending. Such a selection criterium would violate property *AL3-01*, since a continuous stream of high priority service requests could cause starvation of the low priority service requests. It is therefore important to modify the informal problem statement as follows:

"The order in which the various service request categories are served is based on some application-dependent criteria. For example, there may be high and low priority categories and the criterium might be to bias service towards the high priority category. It is, however, essential that the service category selection criteria should ensure that no service request category will be ignored for ever. "

In addition to responsiveness, the call centre example also requires fair responsiveness as indicated in Section 5.4.3.3.

Property *AL3-01* has implications regarding the **reliability** requirements of the system. It implies that if the service request is not aborted by the user and not rejected by the service category allocator, then the service request will be serviced by a service provider. Thus, when the service category allocator assigns a service request to a service queue, it implies that there is at least one active service provider that services service requests of that particular category (from properties *AP1-03* and *AP1-04*). Property *AL3-01* implies that in such a case, at least one of those service providers has to be active or restricted-active at the time when that service request reaches the head of the service queue and is assigned to a service provider. Thus, once a service request is allocated to a service queue (i.e. not rejected by the service queue allocator), the service request will be serviced. It is the function of the service category allocator to check whether at least one element of the appropriate service provider subset is active before it allocates the service request to the service queue.

5.4.3 Precedence properties

The three precedence (until) properties listed in Section 5.2.4 are now considered.

5.4.3.1 AP1-yy (safe liveness)

A safe liveness property deals with a predicate p that needs to hold **continuously** until the corresponding predicate q holds. The informal problem statement is therefore inspected to identify all the requirements that are of this format. The resulting safe liveness properties are as specified below.

Note that the word 'until' in the informal specification of the correctness properties in this section does not imply that the **until** logical relation should be used in the corresponding formal specification of the property. The same applies to the usage of the phrase 'it is ensured' and the **ensures** logical relation. The decision whether the **until** or **ensures** logical relation is more

appropriate is only made when the property is formalised during the design phase, since it is only during the design phase that the SLOOP statements are considered. At that stage it can therefore be decided whether the predicate *q* should be achieved via a single or multiple SLOOP parallel statements.

Property *AP1-01* ensures that the relationship between a connection and a service request remains unchanged while a service request is being processed by the call centre. The remaining properties deal with the path followed by a service request while being processed by the call centre. Properties *AP1-02* to *AP1-04* specify the actions that take place once the service request reaches the head of the input queue. These properties specify that the service request remains in that position in the queue until all the actions as specified by these properties have been performed.

Property *AP1-05* ensures that a service request remains in the input queue until it is assigned to a service queue. In turn, property *AP1-06* ensures that the service request remains in a service queue until it is assigned to a service provider. Finally, property *AP1-07* specifies that the service request remains assigned to a service provider until the required service has been performed.

Properties *AP1-08* and *AP1-09* deal with the safe liveness aspects of progress timers.

AP1-01.A connection is dedicated to a specific service request from the time that the connection is accepted until the connection is terminated by the service category allocator, the user hangs up or the service provider / service provider simulator has finished serving the user.

AP1-02. If the category of the first service request in the input queue has not yet been determined, then it remains uncategorised until its category is determined or the service user hangs up and the service request is removed from the input queue.

AP1-03. If the category of the first service request in the input queue has been determined and there is no active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is rejected by the service category allocator, removed from the input queue and the associated connection is terminated, or the service user hangs up and the service request is removed from the input queue.

AP1-04. If the category of the first service request in the input queue has been determined and there is an active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is removed from the input queue and appended to the service queue associated with that service request category, or the service user hangs up and the service request is removed from the input queue.

AP1-05.A service request remains in the input queue until it is assigned to a service queue, the associated connection is terminated by the service category allocator or until the service user hangs up.

AP1-06.A service request remains in a service queue until it is allocated to an element of the service provider container or until the service user hangs up.

AP1-07. A service request remains assigned to an element of the service provider container until the service provider completes the service and terminates the connection or until the service user hangs up.

AP1-08.If a progress timer has expired and the associated service request is still present in a service queue, then it is ensured that the timer is restarted and a service request progress update is sent to the corresponding service user.

AP1-09.If a progress timer has expired and the associated service request is no longer present in a service queue, then it is ensured that the timer is not restarted and no progress update is sent.

A number of the above properties, e.g. *AP1-01* and *AP1-07*, convey some of the **reliability** aspects of the system.

Property *AP1-01* implies that the communication provider provides a reliable communication service, since there is no option which specifies that the connection can be terminated by the communication service, i.e. it can only be terminated by the service user, the service category allocator or the service provider / service provider simulator.

Property *AP1-07* implies that a service provider will always service a request to completion, unless the service user aborts the request.

5.4.3.2 AP2-yy (absence of unsolicited response)

The correctness properties in this section provide another example of how the use of a checklist such as the SLOOP checklist of property types could lead to the explicit specification of many aspects of the behaviour of a system. In this case the problem statement indicates that the service requests in the service queues are serviced by service providers. Thus, it specifies what **should** happen. This behaviour is described by property *AP1-06*. However, in addition to this, property *AP2-01* ensures that service requests that are not enqueued in service queues cannot be assigned to service providers. Thus, it **also specifies what should not happen**. Property *AP2-02* has a similar purpose.

AP2-01.A service request is assigned to an element of the service provider container only if the service request has been enqueued in a service queue and has remained in the queue until it was assigned to the service provider container element.

AP2-02.A service request is allocated to a service queue only if the service request has been enqueued in the input queue and has remained in the latter until it was allocated to the service queue.

The informal problem statement is modified as follows:
"The call centre ensures that each service user is serviced on a first-come, first-served basis within each service request category. **This is the only way in which service requests are serviced, i.e. a service request is never serviced out of turn."**

5.4.3.3 AP3-yy (fair responsiveness)

AP3-01.Service requests are added and removed from the input queue on a First In First Out basis, except in the case where the user aborts a service request, in which case the service request may be removed from anywhere within the input queue.

Property *AP3-01* confirms the choice of the InputQueue class to model the order in which service requests are processed. Service requests are added to the end of the queue and removed from the head of the queue, except where the user aborts a service request, in which case the service request may be removed from anywhere within the input queue. The above property also implies that the relative ordering of the elements of the input queue always remains the same.

AP3-02.For each service queue, service requests are added and removed on a First In First Out basis, except in the case where the user aborts a service request, in which case the service request may be removed from anywhere within the service queue.

The service queues model the fact that service requests are processed on a first-come, first-served basis **within each category**. Service requests are added to the end of a service queue and removed from the head of the queue, except where the user aborts a service request, in which case the service request may be removed from anywhere within the service queue. The above property also implies that the relative ordering of the elements of a service queue always remains the same.

5.4.4 Consolidation

The informal problem statement is updated to rectify omissions and errors as identified during the problem analysis.

The problem statement is now rewritten in a tabular format, showing how the various properties listed above describe the requirements as given in the problem statement. This step forms part of the SLOOP method and it is performed in order to **cross-check** that all aspects of the problem statement are covered and also to verify that each specified property serves a useful purpose. The properties are checked for **inconsistencies** and **redundant** properties are removed.

For example, an earlier iteration[4] of the above properties contained the following global invariant:

AS3-12. A service request is not allocated to a service queue if the service provider subset associated with that service category does not contain at least one active element.

However, while performing the consolidation step, it was noticed that the safe liveness properties *AP1-03* and *AP1-04* include the safety aspect covered by property *AS3-12*. The two safe liveness properties as listed below are therefore sufficient to describe these particular requirements of the system.

AP1-03. If the category of the first service request in the input queue has been determined and there is no active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is rejected by the service category allocator, removed from the input queue and the associated connection is terminated, or the service user hangs up and the service request is removed from the input queue.

AP1-04. If the category of the first service request in the input queue has been determined and there is an active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is removed from the input queue and appended to the service queue associated with that service request category, or the service user hangs up and the service request is removed from the input queue.

The updated parts of the informal problem statement are shown in bold in the table below.

Note that the terms 'service provider' and 'service provider simulator' are used interchangeably in the table below, unless stated otherwise.

[4] For brevity earlier iterations are not shown .

Informal problem statement	**Correctness properties**
A call centre is a non-terminating system which ensures that dial-in users are serviced in a specified order by the available service providers.	*AL3-01*, *AP3-01* and *AP3-02*. The system is non-terminating, therefore no partial or total correctness properties are defined. Instead, responsiveness and fair responsiveness properties describe the way in which the system responds to events.
The service user places a call to a central (usually toll-free) number. Physically, this number may represent several lines (if line hunting is provided by the Public Switched Telephone Network), or it may represent a single high speed line which can carry multiple connections. The number of calls that can be handled **simultaneously** is therefore **bounded**. If the service user receives a ringing tone, it implies that a connection or line is available, otherwise a busy tone is signalled. The call centre is not responsible for controlling the ringing and busy tones. That is performed by the Public Switched Telephone Network (PSTN). **The call centre has a required capacity of maxConn connections, where maxConn is a positive integer. Once a connection has been accepted by the call centre, the associated service request may only be rejected if none of the active service providers have the capability to service the request. The service request may not be rejected due to any other resource problems.**	*AS2-01* to *AS2-03*. These clean behaviour properties model the restrictions on the number of simultaneous connections imposed by the communication provider. They also specify explicitly that the call centre itself imposes no **further** restrictions on the number of simultaneous connections (due to lack of resources) by specifying the capacity requirements of those modelling elements that are affected. *AS2-04* to *AS2-05*. These clean behaviour properties ensure that obsolete entries (i.e. service requests of connections that have been terminated) do not remain in the input queue or service queues. Thus, an entry in one of these queues is always associated with an active connection. *AL3-01*. This property specifies that once a connection is accepted, its associated service request may only be rejected by the service category allocator. *AP1-03*. The conditions for rejecting a service request once the connection has been accepted, are described in *AP1-03*.
The line or connection that is occupied by a service user remains so for the duration of the call, i.e. until the user hangs up, the connection is terminated due to a problem or the service provider has finished serving the user.	*AP1-01*, *AP1-03*, *AP1-07*. *AP1-01* describes the association between a connection and a service request. *AP1-03* specifies that a connection is terminated if the service category allocator rejects the service request. *AP1-07* ensures that a service provider eventually finishes servicing the service request.
The identity of the service user is irrelevant, therefore there is no upper limit to the number of service users, but they cannot all connect to the call centre simultaneously. Since the call centre does not store any information about the service users for switching purposes, there are no physical constraints regarding the maximum number of service users that can be supported.	This is modelled by the fact that there is no container class for the service users.
The call centre processes the calls on a first-come, first-served basis, i.e. as they are received from the PBX.	*AS3-01*, *AP3-01*. *AS3-01* specifies that the establishment of a new connection implies that the associated service request is appended to the input queue. *AP3-01* specifies that this queue is processed on a

	first-come, first-served basis.
If the call centre supports more than one service request category, it receives information about the type of service required by the service user together with the call. This information is obtained from the service user via an Interactive Voice Response (IVR) before the call reaches the call centre. The call centre uses this information to categorise the service required by the service user. If at least one service provider is active which provides a service of the specified category, the service request is accepted, otherwise it is rejected and the connection is terminated.	*AP1-02* to *AP1-05*. *AP1-02* deals with the categorisation of the service request, while *AP1-03* to *AP1-05* describe the requirements for the allocation of a service request to a service queue.
The call centre ensures that each service user is serviced on a first-come, first-served basis within each service request category. **This is the only way in which service requests are serviced, i.e. a service request is never serviced out of turn.**	*AS3-01*, *AP1-05*, *AP3-01*, *AS3-02* to *AS3-04*, *AP3-02*, *AS3-10*, *AP2-01* and *AP2-02*. When a connection is established, the associated service request is entered into the input queue (*AS3-01*), where it remains until it is allocated to a service queue (*AP1-05*). The input queue is a FIFO queue (*AP3-01*). *AS3-02* to *AS3-04* ensure that each service request is allocated to the correct service queue. *AP3-02* specifies that the service requests in each service queue are allocated to service providers on a first-come, first-served basis. *AS3-10* ensures that service requests are assigned to the correct service providers. *AP2-01* and *AP2-02* ensure that service requests are never serviced out of turn.
There are a number of queues (one for each service request category). **The call centre has to support maxCategories of service request categories. All supported service categories must exist after initialisation of the call centre.**	*AS3-02*, *AS3-03*, *AS2-07*. *AS3-02* states that there is a service category associated with each service queue and *AS3-03* specifies that each service category is unique. *AS2-07* is concerned with the number of service request categories that have to be supported.
Each service request is entered into one of these service queues and each queue is serviced on a first-come, first-served basis. **The call centre ensures that no service request is duplicated or lost.**	*AS3-04* to *AS3-06*, *AS3-09*, *AP2-02*, *AP3-02*. *AS3-04* to *AS3-06* deal with the prevention of duplicate service requests, while *AS3-09* is concerned with the possibility of service requests that could get lost. Since the previous four properties are based on the assumption that service requests reach service providers via service queues, two additional properties are required, viz. *AP2-02* and *AP3-02*. The former ensures that a service request cannot reach a service provider unless it has been enqueued in a service queue and AP3-02 ensures that the service requests in each service queue are allocated to service

	providers on a first-come, first-served basis.
The call centre ensures that service requests of one category are not affected by inadequate resources allocated to service requests of another category.	*AS2-03*.
Once the service request has been allocated to a service queue, the service user is informed at regular intervals of his/her progress towards being served.	*AS3-07*, *AL2-01*, *AP1-08* and *AP1-09*. *AS3-07* ensures that the progress timer is started when the service request is entered into a service queue. *AL2-01* states that a timer which has been started will eventually expire or it will be stopped. *AP1-08* specifies that the service user receives a progress update and the timer is restarted if the service request is still present in the service queue when the progress timer expires. *AP1-09* describes the behaviour if the service request is no longer present in the service queue when the timer expires.
If there is only one service request category, there is no need to obtain information from the service user in order to categorise the service required.	Since this information is obtained from the service user before the connection is processed by the call centre, no properties need to be specified to reflect this behaviour.
The order in which the various service request categories are served is based on some application-dependent criteria. **For example, there may be high and low priority categories and the criterium might be to bias service towards the high priority category. It is, however, essential that the service category selection criteria should ensure that no service request category will be ignored for ever.**	*AL3-01*. This property specifies that a service request will eventually be serviced, unless it is rejected or aborted.
There are one or more service providers. The number of service providers is bounded. **The call centre has to support maxSP of service providers. All supported service service providers must exist after initialisation of the call centre. In the case where the service providers are simulated, the service providers are replaced by maxSP of service provider simulators.**	*AS2-08* and *AS2-09*. These properties specify the number of service providers / service provider simulators that need to be supported. The fact that the number of service providers is bounded, implies that a service request may not necessarily be serviced immediately once it has been categorised. This is modelled by the fact that it is enqueued in a service queue and each service queue is required to have space for at least `maxConn` service requests (*AS2-03*).
Each service provider may service one or more service request categories. For example, one type of service provider (say the elementary type) may service queries only. Another type of service provider (say the advanced type) may be able to service both queries and other transactions. There are therefore one or more service provider categories.	*AS3-08*, *AS3-11*. The fact that a ServiceProviderCategories instance could contain multiple elements (*AS3-08*) implies that there can be one or more service provider categories. Each service category has a service provider subset associated with it. Property *AS3-11* states that a service provider may be an element

	of multiple service provider subsets simultaneously, which implies that a service provider may service one or more service request categories.
Each service request category should be serviced by at least one service provider category.	*AS2-10*, *AS3-08*. The combination of these two properties implies this requirement.
The service request categories and the service provider categories referred to in this problem statement are devised in order to provide flexibility in the way in which service requests are allocated to service providers. These categories may therefore be independent of any other service request categorisation performed by the service providers once a service request is allocated to one of them.	The categorisation that may be performed by the service providers is beyond the scope of the call centre and is not specified here.
Once a service request has been categorised and enqueued in a service queue, the call centre ensures that it is eventually allocated to a service provider, unless the service user hangs up beforehand. The nature of the service being provided is irrelevant to the call centre.	*AP1-06*, *AP1-07*, *AS2-06*. *AP1-06* specifies that once a service request is entered into a service queue, it remains there until the connection is terminated or until it is assigned to a service provider. *AP1-07* ensures that a service provider eventually finishes servicing the service request. *AS2-06* ensures that a service provider becomes free once it has finished servicing a service request.
The connections between the service providers and the call centre may be via the PSTN or they may be established internally via the local PBX.	The communication mechanism between the call centre and the service providers forms part of the environment of the call centre and need not be specified here.
The initial product needs to be tested by simulating the actions of the service users, service providers and the communication provider.	*AL2-02*, *AL2-03*. The simulation of the communication provider and service providers rely on simulation timers to trigger events.
System boundaries: The service providers, service users and communication service form part of the environment of the call centre, i.e. the latter performs a queuing and switching function only. The service providers contain the software to process a service request. This software may vary from application to application and therefore does not form part of the call centre. The service providers may even be supplied by a third party. The software that needs to be designed is restricted to the part pertaining to the call centre and its interface with its environment. This is the part that remains unchanged between applications.	*AS2-01*, *AS3-01*, *AS3-08*, *AS3-09*, *AS3-11*, *AS4-01*, *AP1-07*. These properties specify the requirements that have to be satisfied by the objects in the environment of the call centre. *AS2-01* specifies the requirement that the communication provider should ensure that the maximum number of connections supported by the call centre is not exceeded. *AS3-01* states that the service request associated with a new connection has to be appended to the input queue at the time when the new connection is established. *AS3-08* ensures that each element of the service provider subset belongs to one of the categories in the ServiceProviderCategories instance associated with that service category. *AS3-09* requires that a service provider should not service more than one service request at a time. *AS3-11* specifies that a service provider may be an element of

	multiple service provider subsets simultaneously. *AS4-01* implies that an active service provider subset element may not become inactive if it is the only active element in that subset and the associated service queue is non-empty. *AP1-07* specifies that a service provider will complete the service requested by a service request once the latter has been assigned to it. It also specifies that it will terminate the connection once it has completed the service.
Assumptions: At this level of abstraction a reliable communication medium is assumed between the service users and the call centre as well as between the call centre and the service providers.	*AP1-01.* Failure of the communication medium is not listed as a reason for terminating a connection. This is based on the assumption that the communication medium is reliable.
It is assumed that if a service provider is available at the time when a service request is allocated to a service queue, but subsequently needs to go out of service, it remains available in a restricted fashion until that service request has been processed. The restriction is that it will not process any service requests that were added to the service queue after the service provider had made its intention to go out of service known. A service provider that operates in such a restricted mode may go out of service before the specified service request has been serviced only if at least one other service provider that can provide the specified service is available.	*AS4-01*, *AL3-01.* Property *AS4-01* specifies the required behaviour of the service provider, while property *AL3-01* describes the guarantee that a service request that has been enqueued in a service queue will be serviced unless the service user aborts the request.

The correctness properties specified in Sections 5.4.1 to 5.4.4 describe the desired behaviour of the system as a whole. Some of these properties result from the collaboration of a number of classes, whereas others can be derived from the correctness properties of an individual class. For example, property *AL3-01* requires the collaboration of a number of classes. It states:
Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of a service provider subset, or the service category allocator will reject the service request or the service user will hang up beforehand.

This property requires the ServiceCategoryAllocator instance to process a ServiceRequest instance. It also requires a ServiceCategory instance to assign the ServiceRequest instance to the appropriate ServiceProvider instance eventually.

On the other hand, property *AS3-03* pertains to the ServiceCategory class only. It states:
Each service category is unique.

The next section deals with the specification of the correctness properties of individual classes that form part of the system under development.

5.4.5 Informal specification of correctness properties of individual classes

After the correctness properties of the system as a whole have been specified, the next step of the analysis phase is to identify the correctness properties of the individual classes. The properties of the CommsProviderSimulator and ServiceProviderSimulator classes are shown here as examples. These particular classes were chosen as examples, because they have some properties in common. During the **design** phase these **common properties** are used to determine whether the design can be improved by extracting a **common superclass** from these classes. This aspect of the SLOOP method is discussed in Chapter 6, Section 6.2.3.

The class properties eventually result in the **identification** of the **methods** of these classes during the design phase. During the design phase the properties of the individual methods are also specified. Thus, the specification of class properties deals with the population of the *properties-section* within each class definition. The purpose of the **class properties** is to specify the behaviour of the **class and its instances**. It describes the collective behaviour of the methods of the class and its instances at a high level of abstraction. Once the individual methods are defined during the design phase, their individual correctness properties are specified in the *properties-section* of each method. The only method correctness properties that are specified during the analysis phase, are those of the instance creation methods. This is to facilitate the specification of the initial state of an instance.

The **numbering scheme** for class properties is the same as for system properties, except that the property identification numbers are suffixed with the class name in brackets. The class name may be omitted if it is clear from the context which class the properties refer to. The numbers used within each class are independent of the numbers used for the system properties. The numbers of the correctness properties that are specified within the *properties-section* of each method of a class are unique within that class, not within each method. Thus, for each new method that is specified for a class, the correctness property number follows on from the number of the last property in the same category that was specified for that class or one of its methods.

If a correctness property is **overridden** in a subclass, then the modified property in the subclass is identified by the same number and class name suffix as the property in the superclass. For example, if property *DL1-05(CC_Activation)* of the CC_Activation class is overridden in its subclass, CC_SimulationActivation, then the property number in the CC_SimulationActivation subclass remains *DL1-05(CC_Activation)*. The fact that the suffix of a property refers to a superclass indicates to the reader that the property is overriding a superclass property. Any correctness property that is inherited **unmodified** by a subclass is not repeated in the list of correctness properties defined for the subclass.

If a class is used as a building block in a larger system, then it must be ensured that the class properties are **not in conflict** with any of the correctness properties of the larger system. However, it must be possible to determine the behaviour of a class and its instances solely by inspecting the correctness properties of the class and its methods, i.e. it should not be necessary to refer to the properties of any system in which the class might be used in order to understand the behaviour of the class. For example, when the TimerServices class definition is inspected, it should be clear from its correctness properties what services its instances will provide and what would be expected from its clients. This makes it possible to reuse the TimerServices class in many different systems, not only in a call centre system.

The correctness properties that are specified during the **analysis** phase are formulated in terms of the logical relations that are defined for the SLOOP computational model. The only exception is the **results-in** logical relation, which is used in the total correctness properties of the sequential instance creation methods. These methods are used to define the initial state of an object after instance creation.

5.4.5.1 The CommsProviderSimulator class

The purpose of the CommsProviderSimulator class is to simulate events from the communication provider. These events are the connection attempts by service users and they may occur at random intervals. Thus, from the time that a CommsProviderSimulator instance is created, it needs to try and establish new connections to the call centre at random intervals. The random intervals are modelled by starting a random timer after each event has been generated and the generation of the next event when the timer expires.

The behaviour of the simulator **should not violate any of the correctness properties specified for the call centre as a whole**. In particular, it has to ensure that the capacity restrictions are not violated (property *AS2-01*) and that a service request entry is made into the input queue when a new connection is established (property *AS3-01*). This is evident from the table in Section 5.4.4, which shows that properties *AS2-01* and *AS3-01* apply to the CommsProviderSimulator class in particular. System properties *AL2-02* and *AL2-03* refer to simulation timers and simulation events, therefore they are also applicable to the CommsProviderSimulator class.

The correctness properties below describe the behaviour of the CommsProviderSimulator class. As stated previously, the behaviour of each class should be fully specified via its correctness properties. Thus, it should not be necessary to refer to the system properties for the specification of certain aspects of the behaviour of a class. For this reason it might be necessary to duplicate some of the properties specified for the call centre system in the list of properties specified for the class. That would be case when a property specification in the class definition does not need to add further detail (e.g. properties *AL2-01* and *AL2-02* of the CommsProviderSimulator class). In other cases new properties (such as *AP1-01* and *AP1-02* of the CommsProviderSimulator class) are specified that describe the specific role of the CommsProviderSimulator with respect to the behaviour specified for the system as a whole.

AL2-01 (CommsProviderSimulator). When a simulation event is required, a simulation timer is eventually started .

AL2-02 (CommsProviderSimulator). When a simulation timer expires, the simulation object eventually generates an event .

AP1-01 (CommsProviderSimulator). If an event has to be generated and the maximum number of connections have not yet been established, the communication provider simulator ensures that a new connection is established, the associated service request is appended to the input queue and a new communication provider simulator event is again required.

AP1-02 (CommsProviderSimulator). If an event has to be generated and the maximum number of connections have already been established, the communication provider simulator ensures that the event is cancelled and a new communication provider simulator event is again required.

The following total correctness property is specified for the instance creation method of the CommsProviderSimulator class:

AL1-01(CommsProviderSimulator). Instance creation results in the initialization of the instance variables of the CommsProviderSimulator class and its superclasses and in a communication provider simulation event being required.

The CommsProviderSimulator class ensures that system property *AS2-01* is satisfied by adhering to property *AP1-02* (CommsProviderSimulator), while system property *AS3-01* is preserved via property *AP1-01* (CommsProviderSimulator).

Properties *AL1-01*, *AL2-01*, *AL2-02*, *AP1-01* and *AP1-02* of the CommsProviderSimulator class ensure that the cyclic nature of the system is achieved.

5.4.5.2 The ServiceProviderSimulator class

The purpose of the ServiceProviderSimulator class is to simulate the actions of the service provider. The latter has to process the service requests from the service users. The time it takes to process a service request may vary. Thus, once a service request has been assigned to a service provider simulator, it needs to simulate the random period it takes to service the request. The random period is modelled by starting a random timer when the service provider simulator detects that a service request has been assigned to it. Once the timer expires, the connection is terminated.

The behaviour of the ServiceProviderSimulator class should not violate the correctness properties specified for the call centre as a whole. The table in Section 5.4.4 shows that properties *AS3-08*, *AS3-09*, *AS3-11*, *AS4-01* and *AP1-07* apply to the ServiceProviderSimulator class in particular. Properties *AL2-02* and *AL2-03* refer to simulation timers and events, therefore they are also applicable to the ServiceProviderSimulator class. For brevity, the term 'simulator' will be used in the remainder of this section when referring to a ServiceProviderSimulator instance.

AS3-08 ensures that each simulator in a service provider subset belongs to one of the categories in the ServiceProviderCategories instance associated with that service category. *AS3-09* requires that a simulator should not service more than one service request at a time. *AS3-11* allows a simulator to be an element of multiple service provider subsets simultaneously. *AS4-01* implies that an active / restricted-active simulator may not become inactive if it is the only active / restricted-active element in a service provider subset and the associated service queue is non-empty. *AP1-07* specifies that a simulator will complete the service requested by a service request once the latter has been assigned to it. It also specifies that it will terminate the connection once it has completed the service. Property *AL2-02* states that a simulation timer is started when a simulation event is required. Upon expiry of the simulation timer, a simulation event is generated, as specified in property *AL2-03*.

The properties that describe the behaviour of the ServiceProviderSimulator class are now listed, followed by a description of how these properties relate to the previously mentioned properties of the call centre as a whole, viz. *AS3-08*, *AS3-09*, *AS3-12*, *AS4-01*, *AL2-01*, *AL2-02* and *AP1-07*.

AS3-01(ServiceProviderSimulator). A service provider simulator services a single service request at a time.

AS4-01 (ServiceProviderSimulator). If the service provider simulator is active or restricted-active, then it remains active or restricted-active unless an empty service queue is associated with each service category which has this service provider simulator as an element of its service provider subset or the service provider subsets of each of these service categories contain other service provider simulators that are active or restricted-active.

AL2-01 (ServiceProviderSimulator). When a simulation event is required, a simulation timer is eventually started .

AL2-02 (ServiceProviderSimulator). When a simulation timer expires, the simulation object eventually generates an event .

AP1-01 (ServiceProviderSimulator). The assignment of a new service request to the service provider simulator ensures that a new service provider simulator event will be required.

AP1-02 (ServiceProviderSimulator). If a service provider simulator has to generate an event, it ensures that the service provider simulator terminates the connection currently associated with it and becomes available to service a new service request.

AP1-03 (ServiceProviderSimulator). A service request remains assigned to a service provider simulator until the latter completes the service and terminates the connection or until the service user hangs up.

The following total correctness property is specified for the instance creation method of the ServiceProviderSimulator class:
AL1-01(ServiceProviderSimulator). Instance creation results in the initialization of the instance variables of the ServiceProviderSimulator class and its superclasses and it also results in the registration of the service provider simulator with the relevant service categories.

Property *AS3-01(ServiceProviderSimulator)* ensures that property *AS3-09* of the call centre system is satisfied. Property *AS4-01* is an example of a system property that is now rewritten from the perspective of one of the classes involved. Property *AL1-01 (ServiceProviderSimulator)* describes the initialization of the simulator. System properties *AS3-08* and *AS3-11* are preserved via property *AL1-01 (ServiceProviderSimulator)*.

Properties *AS3-01*, *AL2-01*, *AL2-02*, *AP1-01* and *AP1-02* of the ServiceProviderSimulator together ensure that property *AP1-03 (ServiceProviderSimulator)* is satisfied. The latter is the equivalent of system property *AP1-07*, but with specific reference to the ServiceProviderSimulator class.

In Chapter 6 it is shown how the properties of the above two classes are used to identify a class in the repository with similar properties. The latter is eventually used as a parent class of the above two classes.

5.4.6 Summary of the requirements analysis phase steps

The steps that are performed during the analysis phase are summarised below:

- Analyse the informal problem statement.
- Create the class diagram.
- Determine the system boundaries.
- Specify the required behaviour of the system via informal correctness properties. This is done by considering each type of property in the SLOOP checklist of correctness properties. Inspect the informal problem statement in order to specify correctness properties of each property type in turn.
- Update the informal problem statement if deficiencies are found.
- Consolidate the property specifications. This time it is not the checklist of property types that is used as basis for the inspection of the property specifications. Instead, each sentence of the problem statement is checked to make sure that each aspect is sufficiently covered by the specified correctness properties. Remove redundant properties and add additional properties if necessary.
- Specify the analysis level properties for the classes comprising the system. Ensure that they do not violate any system properties and have sufficient functionality to provide what is required by system.

Iteration between any of the above steps is possible. The procedures described here have been demonstrated in detail in the previous subsections.

5.5 Summary

This chapter has sought to illustrate the feasibility of object-oriented analysis driven by the correctness properties of the system. The static structure of the system is obtained using any of the conventional methods. However, when the **behaviour** is specified, it is described in terms of **correctness properties**.

A case study was used to illustrate the benefits of such an approach. The following advantages were noted:

- **Deficiencies** in the problem statement can be **revealed**.
- The focus is not only on what the behaviour should be, but also on what it should not be. **Error conditions** are therefore considered from the outset.
- It assists the software designer in gaining a **thorough understanding** of the requirements.

The above benefits are evident during the requirements analysis phase, but there are additional advantages that only become apparent during the design phase. These aspects will be discussed in the next chapter, but a brief summary is given below:

- The correctness properties specifying the behaviour of the system under development and its constituent classes are used to **identify** possible matching frameworks / design patterns / classes in the repository of **reusable** SLOOP artifacts.
- Although the specification of the informal analysis phase correctness properties is not trivial, the effort required to translate these properties into formal ones, as well as to formalise the design level refinements, can be saved if appropriate reusable artifacts can be found in the repository. The **correctness properties** are therefore **reused** along with the classes themselves. The **scalability** problem is therefore addressed by taking advantage of the reusability aspect of object-orientation.
- The approach followed during the analysis phase encourages the software designer to think in terms of **properties** that are **universally** or **existentially** quantified over **all statements** in the program instead of focussing on **loci of control**. This paves the way for the design phase, where these properties are refined and the **SLOOP statements** are derived. It prepares the designer to write statements that are based on the computational model described in Chapter 4, thereby promoting **seamlessness** between the analysis and design phases.

The **understandability** problem was also addressed in this chapter. The case study was used as a vehicle to explain how the various correctness types are interpreted and to provide examples of their application. The purpose of the checklist developed in Section 5.2.4 was to assist the software designer in avoiding **underspecification**. The issue of **overspecification** was also addressed. The importance of maintaining a **sufficient level of abstraction** was pointed out. That ensures that multiple design options can be considered during the design phase.

The implications of using the SLOOP method during the design phase is the topic of the next chapter.

CHAPTER 6

DESIGN FROM THE SLOOP PERSPECTIVE

6.1 Introduction

The design phase comprises a number of important steps, viz.

- identifying reusable artifacts in the repository (these may be frameworks, design patterns and/or classes),
- refining the specification,
- identifying class and instance methods based on the correctness properties,
- formally specifying correctness properties,
- deriving SLOOP statements,
- reasoning about the correctness of the SLOOP program,
- possibly creating prototypes and
- improving the design using design patterns.

This chapter covers the first five aspects listed above, while Chapter 7 deals with correctness reasoning. Chapter 8 shows how a SLOOP program, one of the deliverables of the design phase, can be implemented and Chapter 9 elaborates further on the topic of incorporating design patterns.

Multiple iterations over all these steps may be required. In the sections below the issues discussed for each step do not necessarily belong to the same iteration. For example, Section 6.2 describes the identification of reusable artifacts. Some reusable classes may already be discovered during the first iteration, whereas it may be necessary first to perform some design level refinements (the topic covered in Section 6.3) in order to discover some of the other reusable classes. The actions described below are therefore not described in chronological order, but rather according to the topics of identifying reusable artifacts, refining the specification, formally specifying correctness properties and deriving SLOOP statements.

One of the issues that are highlighted in this chapter is the fact that the **reuse** of artifacts stored in a repository **influences** the way in which the properties of design level refinements are reconciled with the analysis level properties. In the SLOOP method this step in the software development process is called the **mapping** of problem domain objects onto solution domain objects, in order to reflect the fact that it is not purely a refinement step.

It will be argued that the design process is not merely a matter of refinement and proving that the refinement preserves the higher level properties. In some cases the reusable component may have additional properties that are not required by the analysis level class, but also do not violate any of the properties of the system under development. In that case the class from the repository can still be used to represent the analysis level class. Furthermore, the reusable class description might contain methods that are not required by the new system and which may or may not

violate the properties of the latter. The SLOOP approach under these circumstances is described in Section 6.2.1.

That section also highlights the impact of placing such emphasis on reusability during the design phase. While trying to map analysis level classes onto classes from the repository, the responsibilities allocated to the various analysis level classes might have to be adjusted in order to take advantage of the reusable classes. This often results in a more elegant solution. Whereas the aim in conventional refinement methods usually is to merely add more detail at each refinement level [BaWr89, Back89, ShLa89], the focus in the SLOOP method is to identify reusable artifacts while adding more detail. As a result adjustments could be made to earlier levels of refinement, provided the system still meets its requirements. This characteristic of the SLOOP method is discussed further in Section 6.2.1.

The **level of formalisation** of correctness properties is another topic that receives attention in this chapter. This is particularly relevant in view of the stated goal of making the SLOOP method **usable by practising software designers.**

Finally, it is shown how SLOOP statements are arrived at and how a SLOOP program is constructed from its constituent classes.

Note that although the discussions in this and other chapters of this thesis usually refer to the actions taken by "the software designer", that does not imply that the SLOOP method restricts one to a single-person approach to software engineering. Since the SLOOP method is an object-oriented method, a system is designed as a set of classes. The modularity of such a system makes it particularly suitable for a **work-sharing multi-person approach** towards software development.

6.2 Identifying reusable artifacts

This chapter continues with the call centre case study. The first step is to try and find one or more frameworks that match one or more subsystems. The class diagram and correctness properties identified during the analysis phase are compared with those of the frameworks in the repository. As noted in Chapter 5, the format of the correctness properties yielded as a deliverable of the analysis phase is informal. The properties of the artifacts in the repository are recorded both formally and informally.

The software designer needs to compare the informal property specifications of the system under development with those of the artifacts in the repository and decide whether they convey the same meaning. The formal specification of the property in the repository is used to confirm that its associated informal description is interpreted correctly.

The question arises whether it would not have been more efficient to have compared **formal** descriptions of properties. There are two reasons for rejecting this option. The first was pointed out in the previous chapter, viz. if a match for the informal property is found, then the **translation** from the informal to the formal format of the property is **reused**.

The second is the fact that even formal descriptions of the same property might **not** be **identical**. For example, different names might be used for the classes, instances and methods referred to by the property. Furthermore, the properties might not be specified at exactly the same level of abstraction. This is particularly true of the property specifications of frameworks, which are likely to be at a higher level of abstraction than that of the specific application being developed. It is the task of the software designer to recognise such differences in levels of abstraction and to take advantage of any reusable artifacts.

This particular problem, viz. finding components in a repository and recognising their applicability, is also discussed in [SiCh97]. It is stated there that "a specification describing a component can be expressed in many equivalent ways". It is also claimed in [SiCh97] that it is **not tractable** to establish the applicability of a component in the repository **automatically**. It is the task of the software designer to determine whether a component in the repository is equivalent to or a refinement of the component defined for the system under development.

As indicated by the flowchart in Figure 4-10(a), if a framework exists which describes the complete system, it is instantiated, i.e. object composition and subclassing techniques are used to adapt the framework to the system being developed. In very large systems, framework(s) might exist for part(s) of the system. In that case the relevant framework(s) are instantiated and the remainder of the system is developed as described below.

If no suitable frameworks exist in the SLOOP repository, the individual classes are considered. The properties that were identified for each class during the analysis phase are used to determine whether appropriate reusable classes can be found in the repository. The class descriptions in the repository are in the SLOOP format. For the purposes of this example it is **assumed** that the **repository** contains a SLOOP description of the `EventSimulator` abstract class, as well as SLOOP descriptions of the Smalltalk library classes.

6.2.1 Mapping problem domain objects onto solution domain objects

It is found that some of the existing generic solution domain classes in the repository, such as the Smalltalk `OrderedCollection` and `Array` classes, have sufficient functionality to represent some of the new classes defined during the analysis phase. For example, the `InputQueue` class could be replaced by `OrderedCollection`, since all the properties identified for the `InputQueue` class during the analysis phase are supported by the `OrderedCollection` class. Before presenting the motivation for this decision, the following must be noted regarding the discussion that follows:

- The discussion of this example is at a detailed level for illustrative purposes.
- In the discussion below the specifications of the properties of the `InputQueue` and `OrderedCollection` classes correspond almost *verbatim*. They only differ when a particular issue is illustrated by the difference. In practice the properties of two classes specified by different authors are likely to be much further apart from each other, both in number and in wording. However, since the purpose of this discussion is to illustrate certain aspects of the task of finding appropriate reusable classes, the examples are written in such a way so as to focus on the issues being highlighted. The discussion is illustrative of the task in broad terms.

Although the identification of the analysis level correctness properties of the `InputQueue` class formed part of the actions discussed in the previous chapter, they were not discussed (for brevity, only the correctness properties of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes were covered there). The analysis level correctness properties of the `InputQueue` class are now described in order to provide the necessary background to the present discussion.

Listed below are all the analysis level call centre correctness properties that **refer** to an `InputQueue` instance. This list is used as the basis for the identification of the analysis level correctness properties of the `InputQueue` class.

AS2-02. The minimum capacity of the input queue is equal to the capacity of the connection container.

AS2-04. If a connection is terminated, it implies that its associated service request is not present in the input queue.

AS3-01. The establishment of a new connection implies that the associated service request is appended to the input queue.

AS3-05. A service request is only present in the input queue if it has not been allocated to a service queue.

AP1-02. If the category of the first service request in the input queue has not yet been determined, then it remains uncategorised until its category is determined or the service user hangs up and the service request is removed from the input queue.

AP1-03. If the category of the first service request in the input queue has been determined and there is no active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is rejected by the service category allocator, removed from the input queue and the associated connection is terminated, or the service user hangs up and the service request is removed from the input queue.

AP1-04. If the category of the first service request in the input queue has been determined and there is an active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is removed from the input queue and appended to the service queue associated with that service request category, or the service user hangs up and the service request is removed from the input queue.

AP1-05. A service request remains in the input queue until it is assigned to a service queue, the associated connection is terminated by the service category allocator or until the service user hangs up.

AP2-02. A service request is allocated to a service queue only if the service request has been enqueued in the input queue and has remained in the latter until it was allocated to the service queue.

AP3-01. Service requests are added and removed from the input queue on a First In First Out basis, except in the case where the user aborts a service request, in which case the service request may be removed from anywhere within the input queue.

When the above properties are analysed, it is clear that the only actions that are performed on the input queue itself, are to **add, remove and access** elements. Many of the properties specify **additional** issues, such as the fact that a new element is appended to the queue **when a connection is established** (*AS3-01*) and the fact that when a service request is removed from the input queue, then **it is appended to a service queue** if the service request is not rejected or aborted (*AP1-05*) . However, this additional information is not relevant to the characteristics of the input queue itself.

The above properties therefore yield the following information regarding the characteristics of the input queue itself :

- The input queue has a **minimum capacity**, which must be equal to the capacity of the connection container (AS2-02).
- Elements are **appended** to the input queue (AS3-01, AP3-01).
- Elements are **removed** from the head of the queue (AS2-04, AP1-02, AP1-03, AP1-04, AP3-01), as well as from any other position in the input queue (AS2-04, AP1-05, AP2-02, AP3-01).
- Elements are **accessed** for the purpose of categorisation (AP1-02).

It is clear that the input queue is responsible for maintaining the **relative** ordering of its elements. However, when deriving the correctness properties of the `InputQueue` class, one has to determine whether the FIFO order in which elements are **added** or **deleted** from an `InputQueue` instance should be the responsibility of the instance or of its clients. Analysis of property *AP3-01* yields the insight that although an `InputQueue` instance could force new elements to be added to the tail of the queue at all times, it should not enforce removal of

elements to be from the head of the queue only. This is because it might be necessary to remove an element from an arbitrary position in the queue if the connection is aborted.

This implies that the responsibility for ensuring that the elements of the queue are processed in a FIFO order should lie with the clients of the input queue. Thus, the `InputQueue` instance should provide methods to add an element to the tail of the queue, remove an element from the head of the queue and remove an element from any position in the queue. However, it is the responsibility of the client to invoke these methods in such a way that elements are added and deleted in a FIFO order under normal circumstances (i.e. when a service request is not aborted). Since the `InputQueue` instance will not enforce the FIFO order, it will therefore not present a problem if the `InputQueue` instance provides a method to add an element to any position in the queue, as will be seen below.

The correctness properties of the `InputQueue` class resulting from the analysis phase should be consistent with the foregoing properties that relate to the characteristics of the input queue. They are identified as follows:

AS3-01 (InputQueue). The elements of the input queue are always in contiguous positions.
AS4-01 (InputQueue). The relative ordering of any two elements of the input queue remains the same unless one of the elements is removed.
AS4-02 (InputQueue). An object that is not an element of the input queue remains not an element unless it becomes an element of the input queue at a specified position.
*AS4-03 (InputQueue). An object that is an element of the input queue remains an element unless the **object** is specified and it is removed from the input queue or the **position** of the object is specified and the corresponding object is removed from the input queue.*
AS4-04 (InputQueue). The capacity of the input queue is non-decreasing (i.e. the capacity of the input queue remains the same unless it grows).

The following is the total correctness property of the `setup` instance creation method of the `InputQueue` class:
AL1-01 (InputQueue). Instance creation results in the capacity of the instance being equal to the capacity of the connection container.

The *AS3-01*, *AS4-01*, *AS4-02* and *AS4-03* `InputQueue` properties together describe the role of the `InputQueue` class in the preservation of property *AP3-01* of the call centre system. Note that the *AS4-02* and *AS4-03* `InputQueue` properties merely allow the client to add and delete elements at specified positions; the values of those positions are the responsibility of the client. The *AL1-01* and *AS4-04* `InputQueue` properties are used to preserve property *AS2-02* of the call centre system.

The above `InputQueue` properties are used during the search for a matching class in the SLOOP repository. The relevant correctness properties of the Smalltalk `OrderedCollection` class are as follows:
AS3-01 (OrderedCollection). The elements of anOrderedCollection are always in contiguous positions.
AS4-01 (OrderedCollection). The relative ordering of any two elements of anOrderedCollection remains the same unless one of the elements is removed.
AS4-02 (OrderedCollection). An object that is not an element of anOrderedCollection remains not an element unless it becomes the last element of anOrderedCollection if its position is unspecified or it becomes an element of anOrderedCollection at a specified position.
*AS4-03 (OrderedCollection). An object that is an element of anOrderedCollection remains an element unless the **object** is specified and it is removed from anOrderedCollection or the **position** of the object is specified and the corresponding object is removed from anOrderedCollection.*

AS4-04 (OrderedCollection). The capacity of anOrderedCollection is non-decreasing (i.e. the capacity of anOrderedCollection remains the same unless it grows).

The `OrderedCollection` class has multiple instance creation methods (e.g. the `new` method creates an ordered collection that has a default capacity of 10, while the `with:` method does the same, but also enters the argument of the method as an element of the collection). The instance creation method that is relevant to the `InputQueue` class is the `new:` method. It has the following total correctness property:
AL1-01 (OrderedCollection). If the capacity specified as the argument of the method is greater than zero, then it results in a new instance that has a capacity equal to the value of the argument.

Properties *AS3-01*, *AS4-01* and *AS4-03* of the `InputQueue` and `OrderedCollection` classes are the same, except for the names of the classes. Property *AS4-02 (OrderedCollection)* is an example of a case where the correctness property of the class found in the repository is **not exactly the same** as that of its counterpart in the system under development. However, there is nothing in the `OrderedCollection` property which violates the requirements of the call centre system. More specifically, it does not violate property *AP3-01* of the call centre system. Even though the postcondition of the `OrderedCollection` property is weaker than that of the `InputQueue` property (the former has an additional *or* clause), this property is still acceptable because it does not violate the system requirement to add elements to the tail of the queue.

Property *AL1-01* of the `OrderedCollection` class guarantees that an instance will have a certain specified capacity immediately after its creation. The corresponding `InputQueue` correctness property specifies that the capacity of an instance will be the same as the capacity of the connection container (in the call centre system). Property *AL1-01 (InputQueue)* implies that it is the responsibility of the `InputQueue` class to ensure that the capacity of an `InputQueue` instance is the same as that of the connection container (in the call centre system). By explicitly referring to the capacity of the connection container in the property of the `InputQueue` class, it immediately rules out a design which reuses a generic collection class such as the Smalltalk `OrderedCollection` class. This suggests that the **distribution** of the **responsibilities** allocated to the various classes during the analysis phase should be modified.

Note that such a modification is not required because the analysis level properties are incorrect, but because a **reusable** class was found which represented **a more elegant solution** to the problem. Note also that there is no modification to the system properties of the analysis phase. It merely represents a different distribution of responsibilities amongst the **constituent** classes of the call centre system.

Property *AL1-01 (InputQueue)* is modified as indicated below in order to reflect the shift in responsibilities:
AL1-01 (InputQueue). Instance creation of the InputQueue class results in the capacity of the instance being equal to the specified capacity.

The modification to this analysis level correctness property of the `InputQueue` class now enables one to map the problem domain `InputQueue` class successfully onto the solution domain `OrderedCollection` class that is found in the SLOOP repository of reusable artifacts. This example demonstrates how the **emphasis on reusability influences the refinements** that are performed at each stage of the software development process. Thus, it is not merely a matter of taking the analysis level correctness properties and adding more detail during the design phase. Instead, it is a case of determining whether the reusable classes found in the repository could possibly provide some of the required functionality and then adjusting the distribution of responsibilities amongst the classes of the call centre system accordingly **if necessary**. It is of course possible that reusable classes could be found that add design level detail only, which would make it a case of pure refinement. In the other case, i.e. where the match is not exact, the

reusable class still represents a refinement, since it also contains design level detail such as the algorithms to implement its methods.

This example highlights the fact that the reusable class may contain many methods that are not necessarily required by the system under development. Some of the **individual** methods may contain correctness properties that are in conflict with the correctness properties of the system. For example, the `new` method of the `OrderedCollection` class creates an instance that has a capacity of 10 elements. This implies that the client which sends the `new` message to the `OrderedCollection` will not be able to ensure that the capacity is equal to that of the connection container. However, the important issue here is that **none** of the properties that are defined at the **class/instance level** of the reusable class should be in conflict with the correctness properties of the system. It is the responsibility of the system designer to ensure that any **method** that is used always satisfies the system properties.

There are several ways of addressing this last issue. One option is to create a subclass of the reusable class and to prohibit the use of any of the methods that do not satisfy the system properties. However, this requires a great deal of effort to be spent on studying the properties of methods that might be totally irrelevant to the system under development. The approach that is followed in the SLOOP method is to place the emphasis on the methods that **are** to be used. This is done by explicitly listing all the methods that are being used by the new system in the *partial-class-methods-section* of the appropriate *partial-class-description* of the SLOOP program. It suffices to ensure that the correctness properties of these methods do not violate the system requirements.

The `InputQueue` example has illustrated why this step of the SLOOP method is not called a refinement step. It is better categorised as a **mapping** of problem domain classes onto solution domain classes, since it is not restricted to refinement only.

The other container classes in the call centre system are also mapped onto Smalltalk-80 library classes found in the SLOOP repository. The way in which one arrives at each of those mappings is similar to the procedures followed for the `InputQueue` class as discussed above. The table below summarises these mappings and their justifications.

Analysis level class	**Design level class**	**Justification for mapping**
`ConnectionContainer`	`Array`	The purpose of the `Connection= Container` class is to model the capacity constraints of the call centre system. It is used to indicate that the call centre can handle a maximum of `maxConn` simultaneous connections. At the same time it has to be guaranteed that a minimum of `maxConn` simultaneous connections can be processed. Property *AS2-01* captures these constraints by specifying that the the capacity of the connection container is **equal** to `maxConn`, where `maxConn` is a positive integer. The capacity is therefore a **fixed** value which may neither be decreased nor increased. Unlike the `OrderedCollection` class,

		which increases its capacity if a new element has to be added and it has already reached its full capacity, the capacity of an array is fixed upon instance creation. . The `Array` class is therefore the most appropriate mapping in this situation.
`ServiceCategory= Container`	`Array`	System property *AS2-07* specifies that the size (i.e. number of elements) of a service category container is equal to `maxCategories`, where the latter is a positive integer. The Smalltalk `Array` class is again the most appropriate, since the capacity of an `Array` instance is fixed.
`ServiceProvider= Container`	`Array`	System property *AS2-08* states that the size of the service provider container is equal to `maxSP`, where the latter is a positive integer. The `Service= ProviderContainer` class is mapped to the `Array` class because the capacity of an `Array` instance is fixed.
`InputQueue`	`Ordered= Collection`	As discussed above, the properties of the `InputQueue` class specify that the relative ordering of the elements of an `InputQueue` instance remains the same and that the positions of its elements are contiguous. Its capacity is non-decreasing (i.e. a minimum capacity is specified). The order in which elements are added to and removed from an input queue is the responsibility of its clients. The Smalltalk `OrderedCollection` class satisfies these requirements.
`ServiceQueue`	`Ordered= Collection`	The requirements are similar to those of the `InputQueue` class. Again, the Smalltalk `OrderedCollection` class is an appropriate mapping.
`ServiceProvider= Categories`	`Set`	As in the case of the `InputQueue` and `ServiceQueue` classes, most of the call centre system properties that refer to the `ServiceProviderCategories` instances deal with the behaviour of the elements of these instances. As far as the behaviour of the container class is concerned, there are no requirements for ordering. There are no correctness properties dealing with the capacity of the `ServiceProviderCategories` instances. This is because these instances are populated at start-up and there is no need to cater for dynamic insertion of elements. The Smalltalk `Set` class has the required functionality.

`ServiceProviderSubset`	`Set`	The justification for choosing the Smalltalk `Set` class as substitute for the `ServiceProviderSubset` class is similar to the one given for the `ServiceProviderCategories` class.

Table 6-1. Mappings of problem domain classes onto solution domain classes.

Figure 6-1 shows the class diagram resulting from the mappings in Table 6-1. The class diagram contains the reusable classes that represent initial mappings of analysis level classes onto design level classes. However, these mappings are not necessarily final, as will be evident from Section 6.7. The mapping of the `ServiceProviderSubset` class is revisited when the flexibility and extensibility of the design are considered. Reasons are given in that section for replacing the Smalltalk `Set` class with the Smalltalk `OrderedCollection` class in the `ServiceProviderSubset` mapping.

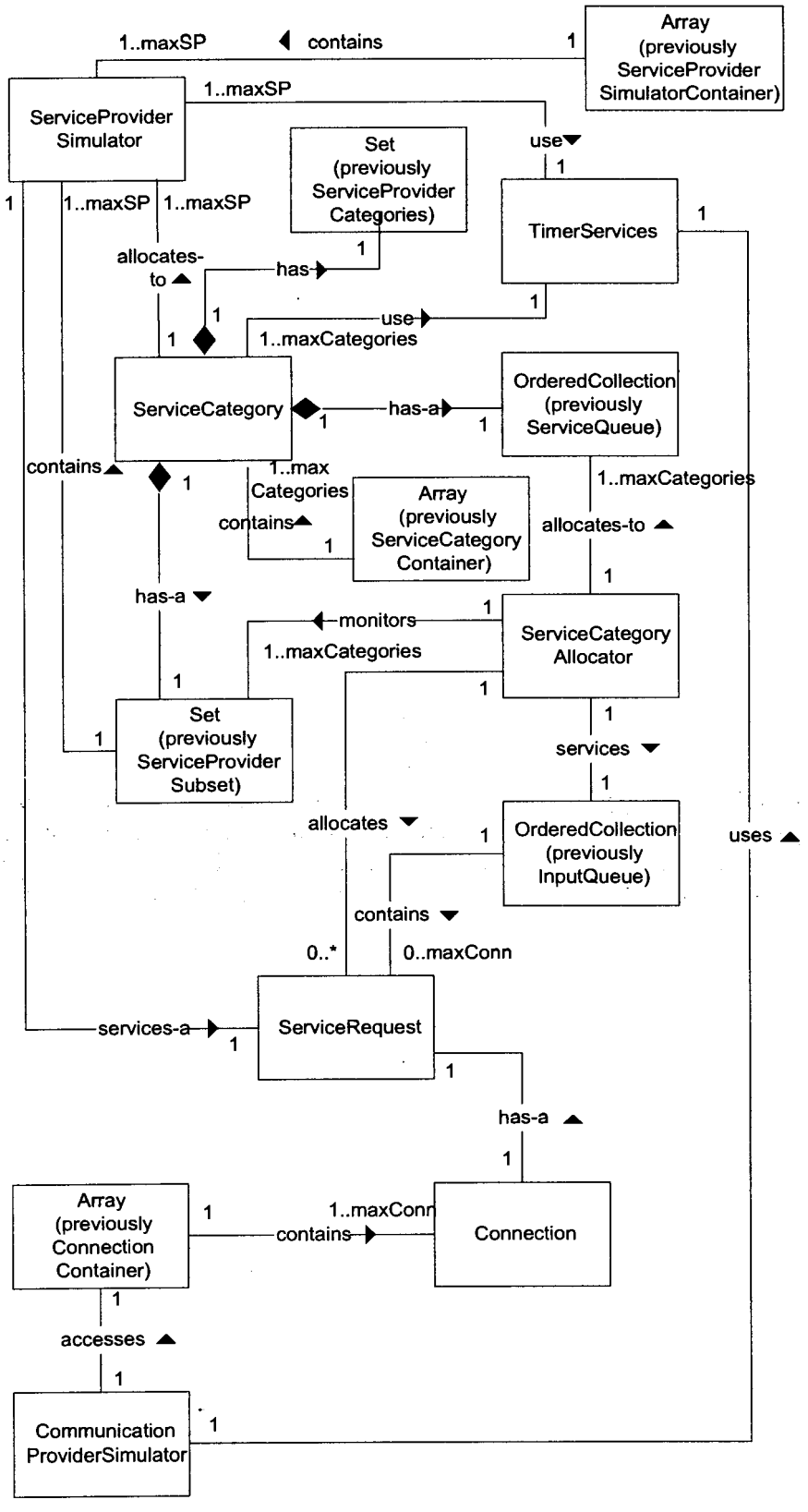

Figure 6-1. Mapping problem domain classes onto solution domain classes.

6.2.2 Formal versus informal specification of class properties

Another issue which is highlighted by the `InputQueue` example in the previous section is the problem of the formalisation of correctness properties at the class level. The problem relates to the fact that some of these properties might refer to aspects that are not represented by relationships between values of attributes of the class, but rather to aspects that are implied by method names or method arguments. The following property of the `OrderedCollection` class illustrates the problem:

AS4-03 (OrderedCollection). An object that is an element of anOrderedCollection remains an element unless the ***object*** *is specified and it is removed from anOrderedCollection or the* ***position*** *of the object is specified and the corresponding object is removed from anOrderedCollection.*

In this property there are references to the specification of an object or a position of an object. Typically, this information would be provided by the client via the argument of the relevant method. However, the argument of a method is not an attribute of the class, i.e. it is not a class or instance variable. The scope of the *pseudo-variable* representing the argument of a method is local to the method itself. It is therefore difficult to write a **class** level property such as *AS4-03(OrderedCollection)* in formal terms. However, the total correctness properties of the individual **methods** can easily be formalised. That is because the *pseudo-variables* are within the scope of the method and can therefore be referenced in the correctness properties of the method.

Similar problems occur when the correctness property of the class refers to aspects that are implied by method names. Again the `OrderedCollection` class serves as an example. It is implied by property *AS4-03* that an element can be selected for removal by specifying its position. However, this position can be implied by the name of the method. For example, the `removeFirst` method implies that the first element has to be removed. Again it is easy to indicate this requirement formally as part of the total correctness property of the `removeFirst` method. However, it is far more difficult to indicate formally in a class level property that clients may specify the position of removal via the name of a method.

It is for this reason that the SLOOP method requires the correctness properties of the individual methods to be specified both formally and informally, while formal specifications in the *properties-section* of the class are desirable, but not mandatory. Examples of formal specifications of method properties are given in Section 6.4.

6.2.3 Reusing existing classes through inheritance

In Chapter 5 the correctness properties of the two simulator classes were given. When searching for a matching class in the SLOOP repository, one first tries to find a reusable class which satisfies **all** the correctness properties of the given class. If such a reusable class is not found, it is still possible that the search might yield one or more classes that satisfy a **subset** of the given correctness properties.

For the purposes of this discussion it is assumed that the SLOOP repository does not contain classes that satisfy all the properties of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes. However, it is further assumed that the SLOOP repository does contain an `EventSimulator` class which satisfies the following correctness properties:

AL2-01(EventSimulator). When a simulation event is required, a simulation timer is eventually started .

AL2-02(EventSimulator). When a simulation timer expires, the simulation object eventually generates an event.

The `EventSimulator` class is an abstract class which provides the functionality to start a timer and to monitor it for its expiry. The conditions that result in an event being required and the actual event that is generated are unspecified, i.e. these aspects are the responsibility of the subclasses. Some additional design level properties regarding the random nature of the timers are also specified.

However, properties *AL2-01* and *AL2-02* of the `EventSimulator` class are the ones that draw the attention of the software designer, because they match properties *AL2-01* and *AL2-02* of the `CommsProviderSimulator` and the `ServiceProviderSimulator` classes. (As in the example in Section 6.2.1, it is unlikely that the properties of the new classes being specified and those of a class in the repository will match almost *verbatim*. The point that is made here, is that these properties are equivalent.)

By making the `CommsProviderSimulator` and `ServiceProviderSimulator` classes subclasses of the `EventSimulator` class, the functionality of the latter is reused. This has the advantage that the software designer only needs to redefine those methods that are left as responsibility of the subclasses.

6.2.4 Discovering suitable existing classes after design level refinements

During the analysis phase it was determined that a `TimerServices` class was necessary in order to provide the timer functionality of the system. While performing the **design level refinements** of the `TimerServices` class, one needs to decide how the timers should be represented and also how the clients of the TimerServices instance should be informed of the expiry of timers. In this example, each timer is represented by an instance of the `TimeoutElement` class. The instance variables (attributes) contain information about the timer, such as a reference to the requestor of the timer, an identifier which uniquely identifies the timer with respect to the requestor and the duration of the timer.

There are several ways in which timeouts can be indicated to the requestors of the timers. One option is to send a message to the requestor (i.e. invoke one of the requestor's methods) indicating that the requested timer has expired. Alternatively, the relevant `TimeoutElement` instance can be entered into a queue of expired timers. Each requestor inspects this queue on a regular basis and if it finds an element representing one of the requestor's timer requests, it removes it from the queue and takes the necessary action.

In this example, the second option is chosen, because it is a more loosely coupled solution. Consider the alternative where the `TimerServices` instance indicates a timeout event by invoking one of the requestor's methods. In that case the `TimerServices` instance is tied up until this object returns control to it if synchronous invocation [Vino97] is used. This is undesirable, because this object may choose to perform several actions as a result of the timeout. Since the timers should be as accurate as possible, the aim of the design is to reduce the influence of other objects on the `TimerServices` instance as much as possible.

As a result of the above design decision, **a new object is introduced**, viz. a queue of expired timers. The correctness properties of this queue are now determined in order to facilitate **comparison** with the classes in the SLOOP repository. The elements of the queue have to be ordered, since the requestors should be able to process their timeout events in the order of expiry. The elements are therefore always added to the end of the queue. However, elements are not necessarily removed from the queue in a FIFO order. For example, consider the case where a requestor is not inspecting the queue for a while (the requestor might be taken out of service

temporarily). When a timeout event for this particular requestor reaches the head of the queue, it will not be removed until the requestor becomes active again. This should not prevent the other requestors from obtaining their timeout information, therefore all requestors are allowed to inspect all elements of the queue and to remove them from any position in the queue.

The properties of the Smalltalk `OrderedCollection` class, first presented in Section 6.2.1, satisfy all of these requirements, therefore this class is reused for the queue of expired timers. This example illustrates how design level refinements could yield new classes, which could then be mapped onto **reusable** classes in the SLOOP repository.

6.3 Refining the specification

The aim of the previous section was to demonstrate how correctness properties are used during the **search for reusable artifacts** in the SLOOP repository. This section deals with the role played by correctness properties during **design level refinements**. It also discusses the impact of the computational model on design level decisions.

6.3.1 The role of correctness properties during design level refinements

As stated in Chapter 5, abstraction during the analysis phase means that design level details are not included. It does not mean that relevant **analysis level details** are omitted. Thus, although the deliverables for this phase may go through a series of refinements, the final refinement of the analysis phase reflects the **complete functionality** of the system. The '**what**' has to be fully specified, while the '**how**' is postponed to the design phase. For example, the first level of refinement might specify normal behaviour only. Error conditions are added in a subsequent refinement. Functionality that is nice to have, but not essential, could be added in yet another refinement. The final analysis phase refinement yields the deliverables for that phase.

During the design phase it is decided **how** the functionality specified during the analysis phase is going to be realised. Again, there may be multiple levels of refinement. In the call centre example the first level of refinement does not include any error conditions (the call centre may not reject a service request and the service user does not abort it). The functionality that is not essential is also not included, i.e. there are no progress timers and the service user is not informed of his/her progress towards being served. The `CommsProviderSimulator` and `ServiceProviderSimulator` classes are included in the design instead of the `CommunicationProvider`, `ServiceUser` and `ServiceProvider` classes.

Design detail is now added for this level of functionality. Two of the important **design** level goals are **reusability** and **flexibility**. The clean behaviour properties in Chapter 5, Section 5.4.1.2, refer to the number of connections, service request categories and service providers that need to be supported by the call centre system. By introducing a `Configuration` class to configure all of the above, the above goals are achieved, since it allows the capacity of the system to differ from application to application, without having to redesign anything for the various applications.

The introduction of a `Configuration` class may not violate any of the properties specified for the system. Part of the behaviour of the `Configuration` class is derived by inspecting the list of system properties. Further design level refinements provide the basis of the remaining part of its behaviour. Listed below are the system properties relevant to the `Configuration` class. Note that these are not properties of the `Configuration` class. They are relevant to the `Configuration` class, because they enable the software designer to determine which aspects of the call centre system should be configurable and what the properties of these configurable items should be.

*AS2-01. The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer.*

AS2-07. The size of the service category container is equal to maxCategories, where maxCategories is a positive integer.

AS2-08. The size of the service provider container is equal to maxSP, where maxSP is a positive integer.

AS3-02. The category of each service request in a service queue is the same as the service category associated with the service queue.

AS3-03. Each service category is unique.

AS3-08. The category of each element of the service provider subset has to match one of the elements of the ServiceProviderCategories instance associated with the service category.

The following design level clean behaviour property results from properties *AS2-01*, *AS2-07* and *AS2-08*:

DS2-01 (Configuration). The values of each of the maximumConnections, maximum= ServiceCategories and maximumServiceProviders[1] *instance variables are invariant and always greater than zero.*

Thus, once the `Configuration` class has been instantiated, the values of the above-mentioned instance variables will be greater than zero and remain constant. The `Configuration` class therefore has to ensure that values greater than zero are assigned to these attributes during system initialization and thereafter the values may be queried, but not modified.

Property *AS3-02* implies that each service request has a category associated with it and this category has to match the category associated with the service queue that it is assigned to. Since these service category names are likely to differ from application to application, it is appropriate to make them configurable items. The `Configuration` class also has to ensure that property *AS3-03* is preserved, i.e. the service category names that are configured for the call centre system have to be unique. The following properties describe the responsibilities of the `Configuration` class regarding the service category names.

DS3-01 (Configuration). The number of service request category names that are configured is equal to maximumServiceCategories.

DS3-02 (Configuration). Each configured service request category name is unique.

Property *AS3-08* implies that a set containing one or more service provider categories is associated with a service category. Again it would be prudent to make the service provider category names configurable items. The `Configuration` class therefore has the following property:

DS3-03 (Configuration). At least one service provider category name is configured.

[1] The names of these instance variables differ from the names used in the system properties. This is because the system properties cannot refer directly to the attributes of one of its classes; it always has to use the methods provided by that class to obtain the values of those attributes. In this example, the maxConn, maxCategories and maxSP macros defined at the system level represent the messages sent to the Configuration instance to obtain the values of the maximumConnections, maximumServiceCategories and maximumServiceProviders attributes respectively.

Note that there is no requirement in the problem statement that each service provider should belong to a unique service provider category. The number of service provider categories that are configured therefore has to be greater than zero, but it does not have to be equal to `maximumServiceProviders`.

The above discussion shows how the `Configuration` class is added during the design phase in order to produce a **flexible** and **reusable** implementation. Although no management functionality is required of the system (based on the the outcome of the analysis phase), this does not preclude a design which makes provision for such functionality, provided the additional functionality does not violate any of the system properties.

When the behaviour of the `Configuration` class is determined, the first step is to inspect the analysis level correctness properties. Subsequently, design level aspects are also considered. In particular, when design level detail is added to the `TimerServices` class, it becomes evident that a maximum allowable timeout value needs to be specified.

This is because the timeout values are represented by positions in a **circular** array[2] and clearly such an array is bounded. Correctness property *DS2-01 (Configuration)* is therefore extended to read as follows:

DS2-01 (Configuration). The values of each of the maximumConnections, maximum= ServiceCategories, maximumServiceProviders and maximumAllowableTimeout instance variables are invariant and always greater than zero.

The `Configuration` class example has served to illustrate how design level refinements can result in the specification of an additional class. In turn, the behaviour of the additional class is determined from both analysis level and design level requirements of the system.

6.3.2 The impact of the computational model on design level decisions

The next topic deals with the impact of the computational model on design level considerations. In the previous chapter it was stated that the purpose of the `ConnectionContainer` class was to model the **capacity constraints** of the call centre. Thus, this class is used to indicate that the call centre can handle a maximum of `maxConn` simultaneous connections. At the same time it also guarantees that a minimum of `maxConn` simultaneous connections can be processed. Property *AS2-01* captures these constraints as follows:

*AS2-01. The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer.*

The above analysis level property does not prescribe whether the elements of the connection container (i.e. the connections) should always be present or whether they should be added and removed as required. It therefore allows for multiple design possibilities regarding the way in which the establishment and termination of connections can be modelled.

One design possibility is to create a `Connection` instance and enter it into the connection container whenever a new connection is established and to destroy it when the connection is

[2] Each position in the array represents a time unit. The TimerServices instance maintains an index into the array. This index is advanced every time unit, taking into account that it is a circular array. Each entry in the array is an ordered collection of TimeoutElement instances. The collection identified by the current value of the index represents timers that have expired during the last time unit. When a new timer needs to be started, a new instance of TimeoutElement is created and it is appended to the collection that will be reached after x time units, where x is the timeout value specified for the new timer. Thus, the appropriate collection is identified by using the value of the index into the array, the size of the array and the requested timeout value in the calculation. For a detailed description of the design of the TimerServices class the interested reader is referred to Section B.11 of Appendix B.

terminated. Another option is to create the maximum number of `Connection` instances upon startup and to use an instance variable (attribute) of the `Connection` instance to model whether a connection is established or terminated.

The SLOOP computational model influences this decision at the design level. When using the SLOOP method, the design is in terms of statements that execute infinitely often. If parallel statements are used to represent the actions of a `Connection` instance, then such an instance has to be present all the time, otherwise one cannot reason in terms of statements that execute infinitely often[3]. For this reason, the design decision is taken to create the `Connection` instances upon startup and to represent the state of the connection via an instance variable.

The SLOOP computational model has another important effect on the design level refinements of a system, viz. the fact that the concept of blocking statements does not form part of a SLOOP design.

For example, the `wait` method of the `Delay` Smalltalk-80 library class causes the active process to be suspended for a specified period. A statement which includes such a message would therefore block for that period. A SLOOP design postpones decisions about assignment of statements to processes to a much later stage of the software development. There is therefore no concept of a suspended process. For this reason the Smalltalk-80 library methods that cause the active process to be suspended, are not used during the design phase.

It is now shown how system services such as timers can be designed within the paradigm of statements that are all executed infinitely often and without using these blocking messages.

One way of implementing timer services using conventional methods is to create a timer services object which suspends itself for a specified period. The value of this period depends on the granularity of the timeouts. Whenever the object resumes execution, it indicates that a time unit has expired by advancing the index into the circular array (the positions in this array represent the timeout values, as described in the previous section). The list of `TimeoutElement` instances at the new position in the array represent the timers that have expired as a result of the expiry of the last time unit. The object then informs the relevant objects. It repeats the cycle of suspending itself and informing other objects of expired timers *ad infinitum*.

In order to make the timeouts more accurate, the above functionality could be distributed over a number of objects, i.e. the object which suspends itself could merely inform one or more other objects that the specified period has elapsed before suspending itself again. The other auxilliary objects could then be responsible for determining whether any timers have expired and inform the relevant objects. This approach would be appropriate if many timers expired simultaneously, resulting in a situation where the time spent between suspended states was no longer negligible.

In the SLOOP environment there is no concept of an object which suspends itself for a specified period. Instead, the `TimerServices` instance contains a parallel method which checks whether the specified period has elapsed and if it has, it advances the index into the circular array and invokes the methods that check whether any timers have expired.

The relevant parts of the parallel statements of the `p_runTimer:` method are shown below (upon instance creation, both `currentTime` and `lastTime` are set to the same value):

```
currentTime := SmalltalkLibPkg:::Time now asSeconds
[] lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 "... and other boolean conditions..."
```

[3] There are exceptions to this rule, as will be discussed in Chapter 9, Section 9.3.2.

The value of `difference` is calculated by evaluating the following *macro-expression*:

```
difference ≡ currentTime - lastTime
                 if (currentTime - lastTime) ≥ 0 ~
               currentTime + (86400 - lastTime)
                 if (currentTime - lastTime) < 0
```

Thus, whenever the statement

```
currentTime := SmalltalkLibPkg:::Time now asSeconds
```

is scheduled for execution, the `currentTime` variable is updated with the latest value of `Time now asSeconds`. The latter provides the number of seconds that have elapsed since midnight.

At some point the other parallel statement, viz.

```
lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 "... and other boolean conditions..."
```

is executed. It calculates whether more than one second has elapsed since the variable `lastTime` has been updated. If it has, the variable `lastTime` is updated with the current value of `currentTime`. The calculation of `difference` takes the rollover at midnight into account. If more than one second has elapsed, the index into the circular array is advanced and the parallel statement that checks whether any timeouts have occurred becomes effective. (This statement is not shown here, but is given in Appendix B, Section B.11. That section contains a detailed description of the internal design of the `TimerServices` class.)

It is clear from the above that SLOOP statements have sufficient expressive power even though blocking messages are not allowed.

6.4 Formalising correctness properties

Once the classes for a particular level of refinement have been identified, it has to be ensured that all the required class and instance methods are defined. At the **method** level it is mandatory to specify the correctness properties formally. This has the advantage of providing an **unambiguous** and **concise** description of the behaviour of the method. The total correctness properties of the `configure` method of the `Configuration` class is now shown as an example. (The purpose of the variables listed below that have not been discussed before, will be given shortly.)

```
"Total correctness property"
"Upon completion of the configure method, the
maximumConnections, maximumServiceCategories, maximumService=
Providers and maximumAllowableTimeout instance variables will
each have a value greater than zero, the srCategoryNames and
spCategoryNames collections will have been created, the number
of elements in the srCategoryNames collection will be equal to
maximumServiceCategories, there will be at least one element in
the spCategoryNames collection, the srToSpCategoryMap will have
been created, the number of mappings in this collection will be
equal to maximumServiceCategories and the categoriesAssigned
variable will have the value zero."
<∀ ( t, u, v, w) where
t > 0 ∧ u > 0 ∧ v > 0 ∧ w > 0 ::
true results-in
      maximumConnections = t ∧
      maximumServiceCategories = u ∧
      maximumServiceProviders = v ∧
```

```
        maximumAllowableTimeout = w ∧
        srCategoryNames notNil ∧ spCategoryNames notNil ∧
        srCategoryNames size = u ∧
        ¬ spCategoryNames isEmpty ∧
        srToSpCategoryMap notNil ∧
        srToSpCategoryMap size = u ∧
        categoriesAssigned = 0
>                                        "DL1-02 (Configuration)"
```

When the properties of a method are specified, one always has to guard against **overspecification**. In this particular example, default values are assigned to all the instance variables. However, the actual values of some of these default values are not presented in the total correctness property, since that would prevent subclasses from assigning other values to those variables. For example, the value of the instance variable `maximumConnections` is set to 8 in the SLOOP statements of this method, while the correctness property merely specifies that it is set to any value `t`, where `t` is greater than zero. This ensures that subclasses can set it to different hard-coded values, or even obtain the values from other sources, while still adhering to the above specification.

The `Configuration` class is a composite class and when it is instantiated, it creates the `srCategoryNames` and `spCategoryNames` objects. These are collections that contain the configured service request and service provider category names respectively. Note that the `srCategoryNames` collection has to contain the maximum number of service category names (indicated by `srCategoryNames size = u`, where `u = maximumServiceCategories`), whereas `spCategoryNames` only has to be non-empty. This is because each service category has to be unique, whereas multiple service providers may belong to the same service provider category.

The `Configuration` class also creates the `srToSpCategoryMap` object. This is a table which maps a service request category name to a collection of service provider category names. This table is interrogated when a `ServiceCategory` instance is created. At that point the `ServiceCategory` instance has to record which service provider categories will be associated with it. The `categoriesAssigned` instance variable is used to keep track of the number of service request categories that have already been assigned to `ServiceCategory` instances. It ensures that each service request category is assigned only once, thereby ensuring the uniqueness of the service request categories.

When the informal and formal versions of the above property are compared, the **concise** and **exact** nature of the formal version becomes apparent. Furthermore, it should be noted that the correctness properties of a method have to contain **all the information** that is required in order to enable the software designer to understand the **impact** of the execution of this method on the **state of the object**. The `configure` method initialises all the instance variables of the Configuration object and this has to be reflected in the correctness property. The ultimate aim of the specification of the method properties is that the software designer should be able to obtain an **exact understanding** of the behaviour of a method without having to study a single program statement.

6.5 Deriving SLOOP statements

Once the classes comprising a system have been identified and correctness properties have been specified for each class, both at the analysis level and the design level, the methods required for each class need to be identified. The class properties are inspected to determine what methods

need to be created. The next step is to specify the correctness properties for each method. Thereafter the SLOOP statements for the individual methods are derived.

The main purpose of this section is to provide an example of how the statements of SLOOP classes are derived. The `ServiceProviderSimulator` class, which is used in this example, also provides a convenient vehicle for highlighting some other issues, viz.

- It is an example of a class which provides its required functionality by simply **complementing** the parallel statements inherited from its superclass with its own parallel statements.
- If there needs to be a relationship at all between the parallel statements of a class and its superclass then that relationship could be via variables rather than via parallel methods that need to be overridden. It is explained why such a design maximises **flexibility** and **extensibility**.
- Criteria for dividing the functionality of a class between its parallel and sequential methods are described.
- The rationale for grouping certain parallel statements into a single rather than into multiple parallel methods is discussed.
- The advantages of using abstract preconditions in SLOOP methods are described.

The `ServiceProviderSimulator` class and its superclass, the `EventSimulator` class, are used as the examples in this section. SLOOP statements are derived for one sequential and one parallel method for each of these classes. However, in order to place the functionality of these methods in context, brief descriptions of the functionality of all the other methods identified for these two classes are required. Summaries of the methods of the `EventSimulator` and `ServiceProviderSimulator` classes are therefore presented in Tables 6-2 and 6-3 respectively[4]. It is beyond the scope of the present discussion to show how these methods were arrived at; only the results of the design phase refinements are shown here.

Note that **abnormal conditions**, such as the user aborting a service request and a service provider simulator going out of service, **are not included at this level of refinement**.

Thus, the abstract `EventSimulator` class provides the common functionality shared by many different types of simulators. It is driven by its parallel method, `p_simulate:timeoutEventsIn:`. The latter sets the `newEventRequired` instance variable to false and invokes `startRandomTimer: withMaximum:`, a sequential `EventSimulator` method, if the `newEventRequired` instance variable is true when the statement starts executing. The `startRandomTimer:withMaximum:` method invokes the relevant `TimerServices` instance method to start a timer.

Message pattern	Discussion
`initialize`	The `initialize` method initializes the instance variables defined for the `Event= Simulator` class.
`startRandomTimer: aTimerServices withMaximum: maximumValue`	This method requests `aTimerServices` to start a timer. The value requested is a random value between 1 and `maximumValue`. (The `startRandom= Timer:withMaximum:` method invokes the `nextRandomNumber:` method to obtain the next random number.)

[4] The detailed SLOOP specifications of these methods are given in Appendix B, Sections B.5 and B.13 respectively.

`nextRandomNumber: maximumValue`	This method returns the next random number within the specified range of 1 to `maximumValue`.
`timerExpired: timerEventQ`	The timerExpired: method returns true if `timerEventQ` contains `aTimeoutElement` representing the expiry of a timer requested by the receiver, otherwise it returns false.
`resetTimerExpired: timerEventQ`	This method removes the `timerEventQ` element representing the expiry of a timer requested by the receiver.
`p_simulate: aTimerServices timeoutEventsIn: timerEventQ`	If `newEventRequired` is true, it invokes the `startRandomTimer:withMaximum:` method and sets `newEventRequired` to false. If the `timerExpired` method returns true, it invokes the `resetTimerExpired:` method and it sets `generatingEvent` to true.

Table 6-2. Methods of the `EventSimulator` class after design phase refinements.

It is the responsibility of the subclass to set the `newEventRequired` instance variable to true. The conditions that result in a new event being required are provided by the subclass, as seen in the `processServiceRequest:` method in Table 6-3 below.

The `p_simulate:timeoutEventsIn:` parallel method also monitors the `timerEventQ` and sets the `generatingEvent` instance variable to true if a timer requested by the receiver has expired and an event therefore has to be generated. The actual event that is generated is again the responsibility of the subclass. The subclass adds its own parallel method to monitor the value of the `generatingEvent` instance variable. This allows for **maximum flexibility** regarding the type of event that is generated by the subclass, since there is no `EventSimulator` method which needs to be overridden by the subclass. The superclass therefore does not even prescribe the name of the method which generates the event, nor does it prescribe the number of arguments that should be passed to it. This aspect of the design is illustrated by the code fragments given below and is discussed further at the end of this section.

The methods of the `ServiceProviderSimulator` are now listed:

Method	**Discussion**
`startSimulation: scContainer using: aConfiguration`	This method creates a `ServiceProvider=Simulator` instance and invokes the `moreInit:using:` method, which performs initialization that is additional to that executed in the `initialize` method of the superclass.
`moreInit: scContainer using: aConfiguration`	The `moreInit:using:` method initializes the instance variables defined in the `Service=ProviderSimulator` class. It also registers the `ServiceProviderSimulator` instance with all the service categories that are serviced by service providers of this category.
`serviceProviderCategory`	This method returns the category of the receiver.
`serviceRequest`	The `serviceRequest` method returns the service request currently being serviced.

`p_generateEvent`	If `generatingEvent` is true, it initiates the termination of the connection associated with the service request, sets `serviceRequest` to nil and sets `generatingEvent` to false.
`registerServiceProvider: scContainer using: aConfiguration`	The simulator registers itself with each service category that requires service from service providers of this service provider category. The simulator keeps a record of all the service categories that it registers itself with. The resulting collection is used to ensure that the simulator does not ignore any of these service categories for ever. This is achieved by maintaining an index into this collection. The index is updated in such a way that the service categories are serviced in a round robin fashion by this simulator.
`processServiceRequest: aServiceRequest`	This method ensures that `newEventRequired` is set to true when a service request is assigned to the simulator. It sets `serviceRequest` to `aService= Request` and it also updates the current index into the collection of service categories being serviced by this simulator.
`canAcceptNextSR: requestingServiceCatgory`	The `canAcceptNextSR:` method returns false if the receiver is busy servicing a request or if the `requestingServiceCategory` does not match the service category that should be serviced next by this simulator, otherwise it returns true.
`p_updateCategoryIndex: scContainer`	This method updates the current index into the collection of service categories serviced by this simulator if the service queue associated with the current service category is empty. This ensures that an empty service queue will not prevent the simulator from servicing the service queues of other service categories.

Table 6-3. Methods of the `ServiceProviderSimulator` class after design phase refinements.

The next step is to **generate the SLOOP statements** of each method. In this example, the SLOOP statements of one sequential and one parallel method from each of the `EventSimulator` and `ServiceProviderSimulator` classes are presented. The interested reader is referred to Appendix B, Sections B.5 and B.13 respectively, for full specifications of these classes. In order to provide the necessary contextual information for these methods, all the class and instance variables, as well as all the class properties are included below.

Note that some of the properties, such as *AL2-01* and *AL2-02*, are only specified informally at the class level. This is because they refer to *pseudo-variables*, a topic which was discussed in Section 6.2.2. However, at the method level all properties are specified formally.

```
class EventSimulator
superclass Object from SmalltalkLibRepository

instance variable names
rand
      "This variable refers to an instance of the Random class from
      the Smalltalk library.  The instance is created when the
```

EventSimulator subclass is instantiated. The instance of the Random class maintains a seed from which the next random number is generated. The random number is used to start a timer with a random value."

newEventRequired

"When the value is equal to true it means that a new event is **required**. Once the variable has been set to true, a random timer will be started at some point afterwards. When the timer is started, newEventRequired is set to false. It is the responsibility of the subclass to set this variable to true when a new event is required, since each subclass will have its own conditions for requiring a new event. Once the timer expires, an event will be generated, as will be described in the comments section of the **generatingEvent** variable."

currentRandomTimeoutValue

"This variable contains the value of the random timeout currently being requested. The purpose of this variable is to provide a mechanism for referencing the current timeout value in the correctness arguments. Note that the SLOOP statements could therefore have been rewritten without this variable while still providing the same functionality. However, in that case it would not have been possible to formalise certain correctness properties (such as DL1-04)."

generatingEvent

"The value is equal to true if the timer has expired and an event has to be **generated**, otherwise it is equal to false. The subclass sets this variable to false at the time when the event is generated. The actual event that is generated is also the reponsibility of the subclass, since each subclass will generate a different type of event."

timerOutstanding

"This variable is set to true when a timer is started and it is set to false when a timeoutElement is removed from the timerEventQ (i.e. when an expired timer has been processed). The purpose of this variable is to provide a mechanism for reasoning about the uniqueness of outstanding timers in the EventSimulator class. In this class only one timer requested by the EventSimulator may be outstanding at a time. The timerOutstanding variable is used in the preconditions of the startRandomTimer:withMaximum: method as well as in the postconditions of the resetTimerExpired: method. If subclasses need to support multiple simultaneous timers, then the preconditions of the startRandomTimer:withMaximum: method need to be weakened and the postconditions of the resetTimerExpired: method need to be strengthened. Since the purpose of the timerOutstanding variable is to facilitate correctness reasoning, the SLOOP statements could have been rewritten without this variable while still providing the same functionality."

timerId

"This variable contains the identifier of the timer currently being requested.

class properties

"Liveness"

"When a simulation event is required, a simulation timer is eventually started."

"AL2-01"

"Liveness"

"If a simulator timer expires, the simulator eventually has to generate an event."

"AL2-02"

"Clean behaviour"
<∀ anObject ::
invariant anObject class ~~ EventSimulator
> **"DS2-01"**
"The EventSimulator class is an abstract class and should not be instantiated"

"Clean behaviour"
invariant rand notNil ∧ rand class = Random **"DS2-02"**
"Once rand has been initialized to refer to an instance of the Random class, it is never set to nil while the instance of the EventSimulator subclass exists. "
"It is therefore possible for the `EventSimulator` subclass instance to send messages to `rand` at any stage after initialization."

"Clean behaviour"
"The currentRandomTimeout value is always within the range specified by the precondition of the start:id:for: method of the TimerServices class."
"DS2-03"

"Global invariant"
"All outstanding timers requested by an EventSimulator subclass instance are identified uniquely with respect to the requestor."
"DS3-01"
"Thus, all the timers requested by this requestor that are currently running or that are in the timerEventQ are uniquely identified with respect to the requestor."

instance methods

category modifying
message pattern startRandomTimer: aTimerServices
withMaximum: maximumValue
"Start a timer with a random value within the range between 1 and maximumValue. When the resulting start:id:for: message is sent to the TimerServices instance, a reference to the requestor (in this case the EventSimulator subclass instance) as well as an identifier are passed as parameters. The combination of the reference to the requestor and the identifier ensures that each timer request can be identified uniquely within the system. This facilitates the correlation of the subsequent timeout notifications with the timer requests.

In the EventSimulator class only one timer is outstanding at a time for a specific requestor, i.e. ¬timerOutstanding is a precondition for starting a new timer for a specific instance of an EventSimulator subclass. Since the timers initiated by a specific EventSimulator subclass instance do not run concurrently, these timers can all have an identifier of 1.

If a subclass requires multiple concurrent timers, unique values must be allocated to the corresponding identifiers. The startRandomTimer:withMaximum: method therefore needs to be overridden in order to achieve this. The total correctness property of the modified method also needs to be updated, viz. a **disjunction** needs to be added to the precondition to state that the proposed identifier of any new timer requested by that EventSimulator subclass instance should not match any identifier of any other outstanding timer requested by that EventSimulator subclass instance. Thus, the precondition has to be **weakened.**

```
        In that case the value of timerOutstanding will no longer be
        relevant."

        method properties
        "Total correctness"
        ¬timerOutstanding results-in methodReturnValue = self ∧
              self postconditions: (#nextRandomNumber:)
              withArguments: #(maximumValue) ∧
              aTimerServices postconditions: (#start:id:for:)
              withArguments: #(self timerId currentRandomTimeoutValue) ∧
              timerOutstanding                                    "DL1-04"
        sequential
        currentRandomTimeoutValue :=
              (self nextRandomNumber: maximumValue)
        [] timerId := 1
        [] aTimerServices start: self id: timerId for:
              currentRandomTimeoutValue
        [] timerOutstanding := true
        end-sequential

category cyclic
        message pattern p_simulate: aTimerServices timeoutEventsIn:
        timerEventQ
        "If a new event is required, start a random timer, the expiry of
        which will cause an event to be initiated."
        method properties
        "This method ensures that properties DS2-03, AL2-01 and AL2-02
        are satisfied by the EventSimulator class."

        "Clean behaviour"
        invariant   currentRandomTimeoutValue > 0 ∧
        currentRandomTimeoutValue ≤ aTimerServices maximumTimeout
                                                                    "DS2-03"
        "A timeout requested by the EventSimulator subclass instance is always within the range
        that ensures that the precondition of the start:id:for: method of the TimerServices class
        is met when the EventSimulator subclass instance invokes that method."

        "Precedence"
        newEventRequired ensures
              self postconditions: (#startRandomTimer:withMaximum:)
              withArguments:
              #(aTimerServices (aTimerServices maximumTimeout))
              ∧ ¬newEventRequired                                   "DP1-01"
        "When newEventRequired is true, it ensures that a simulation timer is started and
        newEventRequired becomes false."

        "Precedence"
        self timerExpired: timerEventQ ensures
              generatingEvent ∧
              self postconditions: (#resetTimerExpired:)
              withArguments: #(timerEventQ)                         "DP1-02"
        "When a simulation timer expires, it ensures that generatingEvent becomes true."

        parallel
        self startRandomTimer: aTimerServices withMaximum:
        (aTimerServices maximumTimeout) \+
        newEventRequired := false
              if newEventRequired
```

```
        [] generatingEvent := true \+
        self resetTimerExpired: timerEventQ
              if self timerExpired: timerEventQ
        end-parallel
```

The `p_simulate: timeoutEventsIn:` method demonstrates very clearly how simple the derivation of the parallel statements is once the correctness properties of the method have been specified. In this case the first and second parallel statements correspond with properties *DP1-01* and *DP1-02* respectively.

The above two methods provide an example of **how the functionality of a class is divided amongst its parallel and sequential methods**. If the class has to react to events, then one or more parallel method(s) are used to monitor those events. The corresponding actions are usually captured in sequential methods. In the above example, either a change to the value of the `newEventRequired` variable or the expiry of a timer constitutes an event. These changes are monitored in the *if* clauses of the parallel statements.

The above example highlights another design issue, viz. the **structuring of parallel statements**. The parallel statements of the `EventSimulator` class are grouped into a single `p_simulate:timeoutEventsIn:` parallel method. Alternatively, each parallel statement could have been encapsulated in a separate parallel method. A single parallel method containing both statements has the advantage that fewer parallel methods need to be activated.

It also ensures that whenever the parallel statement which **starts** a timer is activated, then the corresponding statement which monitors the **expiry** of this timer is also activated. Thus, whenever there is a **dependency** between two parallel statements of the same class, in the sense that if the one is used in a program, then the other should also be used, then it is prudent to group those statements into a single parallel method. That ensures that one of the statements will not be omitted accidentally when the software designer of a new system decides to reuse this functionality[5].

The `ServiceProviderSimulator` class is presented next. The `newEventRequired` and `generatingEvent` instance variables inherited from its superclass are now interpreted with respect to the functionality of the `ServiceProviderSimulator` class. The timer that is started when a new event is required, represents the time it takes to service a service request. When the `generatingEvent` instance variable is set to true, it implies that the service has been completed and the connection should be terminated.

```
class ServiceProviderSimulator
superclass EventSimulator from ApplicationsRepository

instance variable names
serviceRequest
        "This variable refers to the service request currently being
        serviced by the service provider simulator.  Note that the
        reference to the ServiceRequest instance is passed to the
        simulator as a parameter, i.e. the ServiceRequest instance is
        not created by the ServiceProviderSimulator instance and
        therefore does not form part of it."
serviceProviderCategory
        "This variable contains the name of the service provider
        category to which the service provider simulator belongs."
```

[5] Recall that all the parallel methods defined for a class need not be activated when the class is reused. This feature of the SLOOP method was described in Chapter 4, Section 4.3.1.

categoriesServed
"This is an ordered collection containing the names of the service request categories serviced by this service provider. The purpose of this array is to facilitate a round robin servicing scheme of these categories. That prevents starvation of a specific service category."
nrOfCategoriesServed
"This variable contains the number of service request categories serviced by this service provider. It is used in the calculation when the categoryIndex is updated."
categoryIndex
"This variable is used as index into the categoriesServed collection. It is used to determine the next service request category to be serviced by this service provider. It is incremented modulo nrOfCategoriesServed. Its values range from 0 to nrOfCategoriesServed - 1"

class properties

<∀ categoryIndex **where** categoryIndex ≥ 0 ∧
categoryIndex ≤ nrCategoriesServed - 1 ::
invariant serviceRequest notNil ⇒ ¬self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1))
> **"AS3-01 (ServiceProviderSimulator)"**
"A service provider simulator services a single service request at a time."
"If a service request is currently assigned to the simulator, no other service request from any of the categories being served by this simulator will be served by the latter."

serviceRequest isNil ∧ ¬newEventRequired **unless**
serviceRequest notNil ∧ newEventRequired
"AS4-01 (ServiceProviderSimulator)"
"When a new service request is assigned to the service provider simulator then a new service provider simulator event is required."

"Note: The parent class, viz. EventSimulator, contains a parallel method which monitors the value of newEventRequired. If it detects that newEventRequired is true, it starts a timer and sets newEventRequired to false."

serviceRequest notNil ∧ ¬newEventRequired **unless**
serviceRequest isNil ∧ ¬newEventRequired
"AS4-02 (ServiceProviderSimulator)"
"If a service request has been assigned to the service provider simulator and newEventRequired is false, then newEventRequired remains false while the service request is still assigned to the service provider simulator."
"This has the effect that this simulator will not start another timer before the servicing of the current service request has been completed."

generatingEvent ∧ serviceRequest notNil **ensures**
(serviceRequest connection) postconditions: (#terminate:)
withArguments: #('completed') ∧ serviceRequest isNil ∧
¬generatingEvent **"AP1-01 (ServiceProviderSimulator)"**
"If a service provider simulator has to generate an event, it ensures that the service provider simulator terminates the connection currently associated with it and becomes available to service a new service request."

"Note: The parent class, viz. EventSimulator, contains a parallel method which sets generatingEvent to true when a timer has expired."

```
<∀ aServiceRequest where serviceRequest = aServiceRequest ::
 serviceRequest = aServiceRequest ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil
>                          "AP1-02 (ServiceProviderSimulator)"
```

"A service request remains assigned to a service provider simulator until the latter completes the service and terminates the connection."

```
invariant categoryIndex ≥ 0 ∧
      categoryIndex < nrOfCategoriesServed
                           "DS2-01 (ServiceProviderSimulator)"
```

"The categoryIndex is always greater than or equal to zero and less than nrOfCategoriesServed."

```
"Clean behaviour"
invariant    categoriesServed notNil ∧
             categoriesServed class = OrderedCollection
                           "DS2-02 (ServiceProviderSimulator)"
```

"Once categoriesServed has been initialized to refer to an instance of the OrderedCollection class, it is never set to nil while the ServiceProviderSimulator instance exists."

```
<∀ categoryIndex where 0 ≤ categoryIndex ∧
categoryIndex < nrOfCategoriesServed ::
¬(self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1)))
leads-to
      self canAcceptNextSR:
      (categoriesServed at: (categoryIndex + 1))
>                          "DL2-01 (ServiceProviderSimulator)"
```

"For any service category serviced by the service provider simulator, the service provider simulator will eventually be able to service a request from that service category."

```
instance methods
category modifying
message pattern processServiceRequest: aServiceRequest
      method properties
      "A new simulation is required each time when a new service
      request is processed."
      "Total correctness"
      <∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
      categoryIndex = x ∧
      aServiceRequest notNil ∧
      self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
      results-in
            methodReturnValue = self ∧
            serviceRequest = aServiceRequest ∧
            newEventRequired ∧
            categoryIndex = (x + 1) \\ nrOfCategoriesServed
      >                          "DL1-06 (ServiceProviderSimulator)"
      sequential
        newEventRequired := true
      [] serviceRequest := aServiceRequest
```

```
        [] categoryIndex := (categoryIndex + 1) \\ nrOfCategoriesServed
        end-sequential

category cyclic
        message pattern p_generateEvent
        method properties
        "The newEventRequired attribute is not updated here.  It is only
        set to true once a new service request has been received."

        generatingEvent ∧ serviceRequest notNil ensures
                (serviceRequest connection) postconditions: (#terminate:)
                withArguments: #('completed') ∧
                serviceRequest isNil ∧ ¬generatingEvent
                                "DP1-01 (ServiceProviderSimulator)"
        "If a service provider simulator has to generate an event, it
        ensures that the connection currently associated with the
        service request is terminated and that the service provider
        simulator becomes available to service a new service request."
        parallel
        (serviceRequest connection) terminate: 'completed' \+
        serviceRequest := nil \+
        generatingEvent := false
                if generatingEvent
        end-parallel
```

Correctness property *AS3-09* of the system specifies that a service provider / service provider simulator services a single service request at a time. Thus, a new service request should only be assigned to a service provider if it has finished servicing the previous one, i.e. the `processServiceRequest:` sequential method shown above should only be invoked if the `serviceRequest` attribute of the simulator is nil. However, in order to make provision for other conditions that may also play a role when deciding whether a service request can be assigned to a service provider simulator, clients do not interrogate the `serviceRequest` attribute of the service provider simulator, but rather invoke the `canAcceptNextSR:` method. The `canAcceptNextSR:` method facilitates the implementation of an abstract precondition, a concept described in [Meye97].

An abstract precondition is a construct which allows one to strenghten preconditions, without requiring the clients of the target class having to modify their code. Thus, the preconditions of a method are encapsulated in the postconditions of another method (in this example, the `canAcceptNextSR:` method). The client has to ensure that the latter returns true before invoking the `processServiceRequest:` method.

In the above example, the `canAcceptNextSR:` method returns true if the `serviceRequest` instance variable is equal to nil and the service category passed as parameter matches the next category to be served by this `ServiceProviderSimulator` instance. However, in subclasses of the `ServiceProviderSimulator` class, the postconditions of the `canAcceptNextSR:` method might be strenghtened. In this manner the clients of the `processServiceRequest:` method can be sure that the preconditions of the `processServiceRequest:` method will be satisified if they invoke it only if `canAcceptNextSR:` returns true. This remains true even if the **preconditions** of the `processServiceRequest:` method are strengthened (via the strengthening of the **postconditions** of the `canAcceptNextSR:` method). The rationale for including abstract preconditions in a design is to maximise the **extensibility** of the classes. This is an important design level consideration.

The `EventSimulator` and `ServiceProviderSimulator` examples also illustrate how the concept of parallel methods enables one to handle **inheritance** very elegantly in the SLOOP method. In this example the parent class merely sets the `generatingEvent` instance variable to true in the `p_simulate:timeoutEventsIn:` method. It does not invoke the `p_generateEvent` method. This enables the subclass to define any parallel method to act upon the setting of the `generatingEvent` instance variable. For example, the `ServiceProviderSimulator` subclass defines the `p_generateEvent` method, whereas the `CommunicationProviderSimulator` subclass defines the `p_generateEvent: target:` method. (The latter is described in detail in Appendix B, Section B.6.) This approach is possible because the parallel methods are executed infinitely often. The required action will eventually be performed, provided the `generatingEvent` variable is eventually set to the value true and all the necessary parallel methods are activated.

The alternative is to define a `p_generateEvent` method for the `EventSimulator` class and to specify that it is the responsibility of the subclass to implement it. The `p_simulate: timeoutEventsIn:` method invokes the `p_generateEvent` method directly. However, in that case all subclasses are restricted to the selector specified in the superclass. If one subclass requires parameters to be passed to the `p_generateEvent` method, it has to define an additional method and it has to override the `p_simulate:timeoutEventsIn:` method in the parent class to invoke the new method. The first alternative clearly allows the designer more freedom during inheritance.

This section has covered the derivation of the SLOOP statements contained in the SLOOP classes. It is evident from the examples above that **most of the effort** is spent on the specification of the **correctness properties**. Once that has been completed, the derivation of the SLOOP statements is almost automatic, because in most cases there is a very simple correspondence between the correctness properties of a method and the resulting SLOOP statements. Several other issues have also been discussed, all with the aim of highlighting how the SLOOP method aids the designer to generate SLOOP programs that are **modular**, **extensible** and **flexible**.

6.6 Constructing the SLOOP program

Once the classes comprising the system have been identified, the SLOOP program is constructed. The latter consists of an *activation-section* and one or more packages containing the classes used by the system. At this stage it is convenient to define a class which performs the activation function of the system. In the call centre example this class is called `CC_SimulationActivation`. It creates the relevant instances and ensures that the required parallel methods are activated. **The correctness properties of this class are therefore the correctness properties of the system.** (The `CallCentreSimulation` program structure was first presented in Chapter 4, Section 4.3.1, where it can be seen where the `CC_SimulationActivation` class fits into the program.)

The purpose of this section is to **summarise** the results of the actions described in the earlier sections of this chapter **and to show how this information is used to determine the contents of the `CC_SimulationActivation` and `CC_Activation` classes**. This section also demonstrates how an **alternative representation** of the actions performed by the objects of the system can be used to check that the system contains all the required parallel methods to provide the desired behaviour. This alternative representation is given in a format reminiscent of a spreadsheet and its purpose is also to aid **understandability**.

6.6.1 Using the results of the design phase refinements to determine the contents of the sequential methods of the activation classes

The input of the design phase comprises an object model of the problem domain classes and a set of correctness properties describing the required behaviour of the system under development. During the design phase the problem domain (analysis level) classes are mapped onto solution domain (design level) classes. Some of these solution domain classes might be the same as the problem domain classes and others might be different. This depends on various factors:

- **Suitable existing classes** might be found in the SLOOP repository of reusable artifacts (e.g. the `InputQueue` class is replaced by the `OrderedCollection` class).
- Design level refinements might result in the identification of **new classes** (e.g. the `Configuration` class).
- Design level refinements might result in the identification of **new objects**, but the classes of those objects might already exist (e.g. the `OrderedCollection` class can be reused for the `timerEventQ` object).
- Reusable classes could be found that provide functionality that is **common** to some of the analysis level classes. The classes are then restructured to take advantage of the reusable class found in the repository (e.g. the `CommsProviderSimulator` and `ServiceProviderSimulator` classes are modified to inherit their common functionality from the `EventSimulator` class)

The two tables below **summarise** the effects of the design level modifications on the call centre classes and objects. Table 6-4 shows how the design level classes comprising the call centre compare with the ones identified during the analysis phase. Table 6-5 provides a list of all the design level objects defined for the call centre system. The additional objects resulting from the design phase refinements are pointed out. This is followed by excerpts from the sequential methods of the activation classes, demonstrating how the information in these tables is used to determine the contents of these classes.

Design level class	Analysis level class(es)	Relevant repository class
`OrderedCollection`	`InputQueue` `ServiceQueue`	`OrderedCollection`
`Connection`	`Connection`	`Object (superclass)`

`Array`	`ConnectionContainer`
	`ServiceProviderSimulator=`
	`Container`
	`ServiceCategoryContainer`
		`Array`

`ServiceCategory=` `Allocator`	`ServiceCategoryAllocator`	`Object (superclass)`
`ServiceRequest`	`ServiceRequest`	`Object (superclass)`
`Set`	`ServiceProviderSubset` `ServiceProviderCategories`	`Set`
`ServiceCategory`	`ServiceCategory`	`Object (superclass)`
`Configuration`	-	`Object (superclass)`
`EventSimulator`	-	`Object (superclass)`
`CommsProvider=` `Simulator`	`CommsProviderSimulator`	`EventSimulator` `(superclass)`
`ServiceProvider=` `Simulator`	`ServiceProviderSimulator`	`EventSimulator` `(superclass)`
`TimerServices`	`TimerServices`	`Object (superclass)`
`TimeoutElement`	-	`Object (superclass)`
`CC_Simulation=` `Activation`	-	-
`CC_Activation`	-	`Object (superclass)`

Table 6-4. Relationship between the analysis and design level classes.

Table 6-5 shows the instances defined for the classes after the design level refinements have been made (an asterisk indicates that there are multiple instances). In order to ensure that the core classes do not have to be modified when the simulation classes are replaced with the actual interface classes for a specific application, the instance names do not contain any references to the word simulator. Instead, the more generic term "agent" is used.

Design level class	**Instance(s)**

`OrderedCollection`	`inputQ`
	`serviceQ*`
	`timerEventQ` (See *Note 1*)
	`categoriesServed*` (see *Note 2*)
	`timeoutCollection element*` (See *Note 1*)
	`srCategoryNames` (See *Note 3*)
	`spCategoryNames` (See *Note 3*)

`Connection`	the instances* are the elements of the `userConnections` array

`Array`	`userConnections`
	`scContainer`
	`spAgentContainer`
	`timeoutCollection` (See *Note 1*)

`ServiceCategoryAllocator`	`scAllocator`
`ServiceRequest`	`serviceRequest*`
`Set`	`spSubset*` `spCategories*`
`ServiceCategory`	the instances* are the elements of the `scContainer` array
`Configuration`	`config` (See *Note 3*)
`CommsProviderSimulator`	`commsAgent`
`ServiceProviderSimulator`	the instances* are the elements of the `spAgentContainer` array
`TimerServices`	`timer`
`TimeoutElement`	the instances* are the elements of the `timerEventQ`
`CC_SimulationActivation`	`aCCSimulationActivation`
`Dictionary`	`srToSpCategoryMap` (See *Note 3*)

Table 6-5. Classes and instances defined for the call centre case study.

Note 1: The design level refinements of the `TimerServices` class introduced several new objects. The classes of these objects are existing Smalltalk library classes.

Note 2: The `categoriesServed` instance variable of the `ServiceProviderSimulator` class was introduced during the design phase in order to ensure that none of the service queues serviced by a particular `ServiceProviderSimulator` instance would be ignored for ever. After the design level refinements the `ServiceProviderSimulator` class is therefore a composite class containing an instance of the `OrderedCollection` class in order to represent the service categories serviced by the `ServiceProviderSimulator` instance .

Note 3: The `Configuration` class was introduced during the design phase. Its internal design also introduced several new objects. The classes of these objects are existing Smalltalk library classes.

The above information is now used to determine the contents of the sequential methods of the activation classes. The `CC_Activation` class is an abstract superclass which performs the instantiation of all the call centre classes that do **not** form part of the interface, i.e. all the classes that remain **unchanged** whether a simulation or an actual system is running. The methods that instantiate the interface classes are left as the responsibility of the subclasses. The `CC_SimulationActivation` class instantiates the simulation classes.

When an instance of the `CC_SimulationActivation` class is created, the `initialize` method of the parent class, `CC_Activation`, is executed as part of the instantiation. The `initialize` method ensures that all the relevant classes are instantiated upon system startup. In cases where the subclasses should determine which classes should be instantiated, the `initialize` method merely invokes additional methods that can be overridden by its subclasses. In the code excerpt below, the `initCommsAgent` and `initSPAgent` methods are examples of methods that are overridden in the `CC_SimulationActivation` class.

The statements of the `initialize` method are as follows:

```
sequential
config := self initManagement                                    "S1"
[] commsAgent := self initCommsAgent                             "S2"
[] userConnections := SmalltalkLibPkg:::Array new: maxConn
                                                                 "S3"
[] < [] i where 1≤i≤maxConn :: userConnections at: i
      put: (self initConnection: i)
  >                                                              "S4"
[] inputQ := SmalltalkLibPkg:::OrderedCollection new:
maxConn                                                          "S5"
[] scAllocator := self initServiceCategoryAllocator              "S6"
[] scContainer := SmalltalkLibPkg:::Array
    new: maxCategories                                           "S7"
[] < [] j where 1≤j≤maxCategories :: scContainer at: j
    put: (CC_CorePkg:::ServiceCategory setup: config)
  >                                                              "S8"
[] spAgentContainer :=
      SmalltalkLibPkg:::Array new: maxSP                         "S9"
[] < [] k where 1≤k≤maxSP :: spAgentContainer at: k
    put: (self initSPAgent)
  >                                                              "S10"
[] timer:= SystemUtilitiesPkg:::TimerServices setup: config
                                                                 "S11"
[] timerEventQ := SmalltalkLibPkg:::OrderedCollection new
                                                                 "S12"
end-sequential
```

Table 6-6 is used to check that each instance listed in Table 6-5 is accounted for in the above method. In some cases the instances are not specified explicitly in the above statement. However, they are created as a result of classes that are instantiated via the above statements. This is true in the case of composite classes. Some classes need not be created at startup (e.g. the `TimeoutElement` instances). These exceptions are noted explicitly.

Instance(s)	Statement number
inputQ	S5
serviceQ	S8 (the setup: method of the ServiceCategory class results in the creation of a serviceQ object)
timerEventQ	S12
categoriesServed*	S10 (the initSPAgent method results in the creation of a categoriesServed object)
timeoutCollection element*	S11 (the setup: method of the TimerServices class results in the creation of the timeoutCollection elements)
srCategoryNames	S1 (the initManagement method results in the creation of the srCategoryNames object)
spCategoryNames	S1 (the initManagement method results in the creation of the spCategoryNames object)
userConnections element*	S4 (the initConnection: method results in the creation of a userConnections element)
userConnections	S3
scContainer	S7
spAgentContainer	S9
timeoutCollection	S11 (the setup: method of the TimerServices class results in the creation of the timeoutCollection object)
scAllocator	S6
serviceRequest*	S4 (the initConnection: method results in the creation of a serviceRequest object)
spSubset*	S8 (the setup: method of the ServiceCategory class results in the creation of an spSubset object)
spCategories*	S8 (the setup: method of the ServiceCategory class results in the creation of an spCategories object)
scContainer element*	S8
config	S1 (the initManagement method results in the creation of the config object)
commsAgent	S2 (the initCommsAgent method results in the creation of the commsAgent object)
spAgentContainer element*	S10 (the initSPAgent method results in the creation of an spAgentContainer element)
timer	S11
TimeoutElement instances	**These objects are created dynamically after initialization.**
srToSpCategoryMap	S1 (the initManagement method results in the creation of the spCategoryNames object)

Table 6-6. Table used to cross-check that all classes referenced by the system are instantiated appropriately.

As noted earlier, some of the instances listed above are created via methods that are overridden by subclasses of the CC_Activation class. For example, the CommsProviderSimulator class is instantiated in the initCommsAgent method. In the CC_Activation class the statements of this method are as given below:

```
sequential
^self subclassResponsibility
end-sequential
```

This method is redefined as follows in the CC_SimulationActivation subclass of the CC_Activation class:

```
sequential
^CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation
end-sequential
```

The classes that are not created directly via the initialize method of the CC_Activation class are those that are **most likely to be subclassed**. This is an example of the application of the Factory Method design pattern [GHJV95], which will be discussed in detail in Chapter 9, Section 9.3.1. Thus, the application of this design pattern makes it possible to leave the initialize method intact during subclassing. Only methods such as initCommsAgent need to be overridden during subclassing. The interested reader is referred to Appendix B, Sections B.2 and B.3 for details of the other methods invoked by the initialize method (e.g. initManagement).

After initialization, all the class invariants of the instantiated classes have to hold. The progress and precedence properties are achieved via the execution of the selected parallel methods of the various classes in the system (and via the sequential methods invoked by the parallel methods). The next section deals with the topic of ensuring that all the relevant parallel methods of a system are activated.

6.6.2 Determining the contents of the parallel methods of the activation classes

After all the classes have been instantiated during the instance creation of the CC_SimulationActivation class, the parallel methods required by the system need to be activated. This is achieved via the p_activate method inherited from the CC_Activation class.

The Template Method design pattern [GHJV95] is evident in the p_activate method. The design pattern itself and the motivation for its application in the p_activate method will be described in more detail in Chapter 9, Section 9.5.3. At this stage it suffices to say that the main purpose of the p_activate method is to ensure that all the required parallel methods of the call centre system are activated without necessarily invoking those methods directly. As a result many of the statements of the p_activate method are encapsulated in other methods of the CC_Activation class. The statements that are most likely to be overridden are encapsulated. Subclasses may therefore selectively override some of these methods, while the p_activate method remains unchanged. The statements of the p_activate method are given next:

```
parallel
self p_executeCPAgent
"The parallel methods of the commsAgent are not invoked
directly, but rather via the p_executeCPAgent method of the
CC_Activation class."

[] timer p_runTimer: timerEventQ
"Activate the parallel methods of the timer object.  The timer
parallel statements have the following functionality: Whenever a
timeout occurs, the TimeoutElement instance representing the
timeout is added to the end of the timerEventQ, which indicates
to the requestor that the specified timer has expired."
```

```
[] self p_categoriseAndAllocate
"The parallel methods of the scAllocator object are invoked via
the p_categoriseAndAllocate method of the CC_Activation class.
The scAllocator parallel statements have the following
functionality: Once a service request has been categorised, it
is removed from the inputQ and appended to the appropriate
serviceQ."

[] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
      p_execute
  >
"Activate the parallel methods of the ServiceCategory instances.
Their parallel statements have the following functionality: For
each service category the associated service queue and set of
service provider agents are monitored.  If the service queue is
not empty and a service provider agent in the spSubset
associated with the service category is available to process a
new service request, the first element of the service queue is
removed and assigned to a service provider agent."

[] < [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
  >
"The p_executeConnection method of the CC_Activation class is
executed for each Connection instance in order to invoke the
parallel methods of the latter.  The parallel statements of the
Connection instances have the following functionality: When a
connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection.  Once all the procedures have been completed to
terminate the connection, the connection and its associated
service request are reset to their initial states."

[] < [] k where 1≤k≤maxSP :: self p_executeSPAgent:
      (spAgentContainer at: k)
  >
"The parallel methods of the service provider agents are not
invoked directly, but rather by executing the p_executeSPAgent
method of the CC_Activation class for each of the service
provider agents."

end-parallel
```

The `p_runTimer:` and `p_execute` parallel methods of the `TimerServices` and `ServiceCategory` instances respectively are activated directly as can be seen from the statements above. In contrast, the parallel methods of the `ServiceCategoryAllocator`, `Connection`, `CommsProviderSimulator` and `ServiceProviderSimulator` classes are activated indirectly. The statements of the `p_categoriseAndAllocate` and `p_executeConnection:` methods of the `CC_Activation` class activate the parallel methods of the `ServiceCategoryAllocator` and `Connection` classes respectively, as can be seen below:

```
message pattern p_categoriseAndAllocate
method properties
"..."
```

```
parallel
scAllocator p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] scAllocator p_allocate: scContainer from: inputQ
"Once the first service request has been categorised, the
scAllocator removes it from the inputQ and appends it to the
appropriate serviceQ."
end-parallel

message pattern p_executeConnection: aConnection
method properties
"..."
parallel
aConnection p_informCommsProvider: commsAgent
"When a connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection."
[] aConnection p_doWrapUp
"Once all the procedures have been completed to terminate a
connection, the connection and its associated service request
are reset to their initial states."
end-parallel
```

The parallel methods of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes are activated via the `p_executeCPAgent` and `p_executeSPAgent:` methods of the `CC_Activation` class, but these methods are the responsibility of the subclass, as illustrated by the code fragments presented next:

```
message pattern p_executeCPAgent
method properties
"..."
parallel
self subclassResponsibility
end-parallel

message pattern p_executeSPAgent: spAgent
method properties
"..."
parallel
self subclassResponsibility
end-parallel
```

The `CC_SimulationActivation` class redefines these methods as follows:

```
message pattern p_executeCPAgent
method properties
"..."
parallel
commsAgent p_simulate: timer timeoutEventsIn: timerEventQ
[] commsAgent p_generateEvent: userConnections target: inputQ
"The commsAgent simulates the establishment of new connections
at random intervals (within a configured range).  A simulation
timer is started after initialization and restarted each time
after the establishment of a connection has been simulated. The
latter is done by placing the service request associated with
the new connection into the input queue.  The commsAgent ensures
that the capacity of maxConn connections per call centre is not
exceeded, therefore a message is displayed indicating that all
```

```
connections are busy if the maximum number of connections are
currently assigned."
end-parallel

message pattern p_executeSPAgent: spAgent
method properties
"..."
parallel
spAgent p_simulate: timer timeoutEventsIn: timerEventQ
"When a service request has been assigned to a service provider
simulator, the latter simulates the time it takes to service the
service request by starting a random timer.  When this timer
expires, it represents the completion of the service."
[] spAgent p_generateEvent
"When the service provider has completed the service, it
indicates that the connection should be terminated."
[] spAgent p_updateCategoryIndex: scContainer
"Update the index into the categoriesServed collection if the
serviceQ of the current category being served by this spAgent is
empty."
end-parallel
```

Thus, the parallel methods that are specific to the interface classes are activated by methods redefined in the appropriate subclasses of the `CC_Activation` class.

Another way of viewing the dynamics of a SLOOP program is to compare it with the way in which a spreadsheet operates. There are a number of rules and when an event occurs, the rules that are affected are evaluated. This may result in another event, which again results in the evaluation of some rules. This process continues until the spreadsheet reaches a stable state, which is only disturbed if another event occurs.

A SLOOP program comprises a number of parallel statements that execute infinitely often. When an event occurs, the statement which is affected will eventually be executed. As a result of the execution of this statement, other events may be generated. The results of these events will be seen once the affected statements are executed. This process continues until a stable state is reached, i.e when the state remains unchanged no matter which statement is executed.

Table 6-7 below presents the parallel statements and events of the `CallCentreSimulation` program in the spirit of a spreadsheet. Only the parallel statements that appear in the methods of the `CC_Activation` and `CC_SimulationActivation` classes are shown.

Object ---------- Event	comms Agent	timer	scAllocator	scContainer at: j	spAgent Container at: k	user Connections at: i
new comms= Agent simulation timer must be started	p_simulate: timeoutEvents In:					
timer running for comms= Agent		p_runTimer:				
timer for comms= Agent expired	p_simulate: timeoutEvents In:					
comms= Agent must generate event	p_generate Event: target:					
inputQ is not empty and service request not yet categorised			p_categorise: using:			
first element of inputQ categorised, but not yet allocated to a serviceQ			p_allocate: from:			
serviceQ is not empty				p_execute		
service request assigned to a service provider simulator					p_simulate: timeoutEvents In:	
timer running for service provider simulator		p_runTimer:				
timer for service provider simulator expired					p_simulate: timeoutEvents In:	
timer for service provider simulator expired					p_generate Event	
service queue currently being serviced is empty					p_update Category Index	
connection needs to be terminated						p_inform Comms Provider:
connection termination needs to be wrapped up						p_doWrapUp

Table 6-7. Events and actions of the call centre in a tabular format.

The initial trigger is the new `commsAgent` event that must be simulated. It results in a timer being started for the `commsAgent` object (via the `p_simulate:timeoutEventsIn:` method). The `timer` objects monitors all timers via its `p_runTimer:` method. When the `commsAgent` timer expires (detected via a statement in the `p_simulate:TimeoutEventsIn:` method), the `commsAgent` inserts a service request into the `inputQ` (unless all the connections are busy) and indicates that a new event is again required. The whole process is repeated *ad infinitum*.

In the meantime, the `scAllocator` detects that the `inputQ` is not empty. It categorises the first element (via the `p_categorise:using:` method), assigns it to a `serviceQ` and removes it from the `inputQ` (via the `p_allocate:from:` method). This is done whenever the `inputQ` is not empty. Each `ServiceCategory` instance (each one is an element of the `scContainer`) checks the `serviceQ` associated with it. If it is not empty and a service provider can be found that is available and services requests of that category, it assigns the first element in the `serviceQ` to the service provider and removes it from the `serviceQ` (via the `p_execute` method).

The assignment of a service request to a service provider causes a new service provider simulation event to be required. As a result a new service provider simulation timer is started (via the `p_simulate:timeoutEventsIn:` method). The `timer` object monitors all timers (including this one) via the `p_runTimer:` method. When the timer expires, the service provider agent generates an event to terminate the connection (via the `p_generateEvent` method). When a connection has to be terminated, the statements to inform the `commsAgent` are executed (`p_informCommsProvider:`) and the connection is returned to its idle state (`p_doWrapUp`).

In addition to the parallel statements discussed above, the `ServiceProviderSimulator` instances also execute the `p_updateCategoryIndex:` parallel method. This method updates the `categoryIndex` instance variable if the service queue currently being serviced by the `ServiceProviderSimulator` instance is empty.

The sequence in which the statements are executed, is not important. For example, the statement which processes a service queue may be executed before the one that processes the `inputQ`. If there is no service request in the service queue, the former statement will have no effect until the `inputQ` has been processed.

In the case of the `p_activate` method, there are also several quantified statements, e.g.

```
< [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
        >
```

Each instantiation of the above quantification represents a separate statement. Before a statement is selected for execution within a cyclic method, all the quantified statements are instantiated. Only a single statement is selected during an execution of the method. Thus, if a method consists of a quantified statement representing *n* statements only one of those *n* statements is executed when the method is invoked. If a method contains enumerated statements as well as quantified statements, each instantiation of the quantified statement competes on an equal footing with the enumerated statements for selection.

The above tabular representation of the parallel statements of the `CallCentreSimulation` program augments the description of **object interactions** in a SLOOP program. It aids the designer in **checking** that all the parallel statements that need to be executed by the system are indeed activated via the activation class and its ancestors. Once the contents of each class and instance method has been determined, an executable program can be derived. This is the topic discussed in Chapter 8.

6.7 Making the design more reusable

Once a design has reached the stage where it fulfills the functional requirements, it is time to consider improvements in order to make the design more reusable. This aspect of the design phase is covered in detail in Chapter 9, but a brief introduction is provided here.

One way of increasing the **extensibility** and **flexibility** of the design is by considering likely future requirements and by evaluating the ease with which such enhancements could be accommodated. For example, one likely future requirement of the call centre system is the capability to ensure that service requests are not always assigned to the same service provider if multiple service providers are available. This can be achieved quite easily if the `spSubset` of each service category is implemented as an instance of the `OrderedCollection` class rather than as an instance of the `Set` class. It is done as follows: A service request is always assigned to the first element of `spSubset` that is available to accept a new service request. When this assignment is performed, that element is removed from `spSubset` and added to its tail. This ensures that one service provider does not do all the work.

Thus, although the current requirements specification does not prescribe any ordering regarding the allocation of service requests to available service providers, it is prudent to make provision for some type of ordering. The `OrderedCollection` class does not restrict the design to a specific ordering; that depends on the way in which the clients add and remove elements from the `OrderedCollection` instance. However, it does guarantee the relative ordering of its elements while they form part of the collection.

Another way of improving the design is by incorporating design patterns [GHJV95]. An introductory description of an example from the call centre case study is presented next. As stated in Chapter 5, the identification of the service user is important in some types of call centres, because the identity of the service user may play a role in the allocation of the service request to the appropriate service queue. The calling telephone number may suffice as an identification, in which case the above parallel statements need not be changed. However, in other cases it may be necessary to extract some additional information about the service user from a database, using the calling telephone number as key into the database. For example, some service users may be paying an additional fee in order to ensure that all their service requests are treated as high priority requests. Such information may be contained in a database.

Such an extension to the system could easily be accommodated by replacing the algorithm used in the `p_categorise:using:` method of the `ServiceCategoryAllocator` class. Chapter 9, Section 9.5.4, shows in more detail how the Strategy design pattern [GHJV95] can be used to allow for the replacement of one algorithm by another.

6.8 Summary

The purpose of this chapter has been to demonstrate the **feasibility** and **advantages** of the SLOOP approach during the design phase of system development. Feasibility is an important concern, given the goal of making the method **usable by ordinary practising software designers**. The main concern is the amount of effort that is required when placing so much emphasis on correctness properties during the design phase. It was shown that a **pragmatic** approach is advocated in the SLOOP method. Only the **method** properties need to be specified formally. Informal property specifications suffice at the **class** level.

Other considerations also influence the direction taken regarding the level of formality. One aspect is the fact that a formal description of the classes in a system under development might not

match the formal specifications of artifacts in a repository, even though there could be enough similarities to warrant reuse. **Differences in terminology** is one reason for discrepancies.

Another factor is the **incomplete status** of the classes of the new system when the design phase is entered. Quite often design level refinements result in new insights which may even require updates to the requirements analysis phase deliverables. Examples of such situations were given in Section 6.2.1. It is therefore argued that the design process is not a mechanical procedure which could be automated easily, since there are too many factors that require human assessment while searching the repository for reusable artifacts[6] and also while performing the design level refinements. This is one of the reasons why the emphasis is on **informal** property specifications during these activities.

Another reason was the possibility of **reusing** the formal property specifications of the methods of the classes already present in the repository. Once a reusable class has been identified, all the effort of translating informal property specifications into formal ones can be saved by reusing the existing specifications.

Although the SLOOP method is not a formal method, correctness properties play a crucial role throughout the software development process. It was shown how these correctness properties were utilised during various stages of the design phase, e.g. during the identification of reusable artifacts and during the refinement of the design. The **derivation of the SLOOP statements** for each method become straightforward once their correctness properties have been determined. This was illustrated by some examples in Section 6.5. By concentrating on the correctness properties at all stages of the design, the likelihood of **a more correct result** increases.

The **impact** of the **SLOOP computational model** on the design phase was another issue which was investigated in this chapter. Although this model results in certain constraints during the design phase (e.g. blocking statements may not be used), these constraints do not restrict the applicability of the SLOOP method. It is still possible to provide a design for a class such as `TimerServices`, as was shown in Section 6.3.2. In fact, the resulting design has the advantage that issues such as the **assignment of statements to processes** are **postponed** to the implementation phase. The computational model also makes it easy to add new parallel statements during subclassing. As was shown in Section 6.2.3, the concept of **inheritance** fits in very neatly with the SLOOP approach.

The latter part of this chapter demonstrated how a SLOOP program was constructed. Appendix B contains a complete listing of the first level of refinement of the call centre classes. It serves to illustrate the applicability of the method to a **non-trivial** problem.

The next chapter is devoted to **reasoning** about correctness properties. It shows that the correctness properties can be **reused** along with all the other aspects of a class. The implementation phase is covered in Chapter 8, while Chapter 9 shows that the SLOOP design method is **amenable** to the usage of well-known **design patterns**, an issue which was touched upon briefly in Section 6.7.

[6] In [SiCh97] Sivilotti and Chandy also note that it is not tractable to automate the procedures for identifying matching components in the repositories of reusable components.

CHAPTER 7

REASONING ABOUT SLOOP PROGRAMS

7.1 Introduction

The assertions in a SLOOP program serve several purposes. Firstly they provide a means of conveying the **semantics** of the specified classes. Secondly they facilitate **reasoning about the correctness** of the program. Finally, they provide the necessary information to derive the SLOOP statements. All of these aspects are covered in this chapter. It is also shown how correctness reasoning benefits from two key characteristics of the SLOOP approach, viz. **reusability** and the **absence of control flow**.

7.1.1 Conveying the semantics

When a designer seeks information about the behaviour of a class, the assertions associated with a SLOOP class and its methods enable the designer to obtain this information without having to study the code. Section 7.2 describes the **nature** of the information that is obtained from the various parts of a SLOOP class specification.

7.1.2 Reasoning about correctness

Another important role of the assertions is to facilitate reasoning about correctness. Section 7.3 deals with this topic. An example of an informal proof of each **type** of correctness property as listed in Chapter 5, Section 5.2.4, is given.

Each example also serves to cover some **aspect** of correctness reasoning. For example, in one of the discussions a detailed account is given of the **steps** that are usually followed when an informal proof is presented. The use of **induction** [MaPn81b] when proving **invariance** properties and the use of **eventuality chains** [MaPn81b] when proving **liveness** properties are described. Other issues that are covered are, inter alia, the responsibilities of the client of an object and the significance of the `postconditions:` and `postconditions: withArguments:` constructs. Some of the examples presented in Section 7.3 deal with various reusability aspects of correctness reasoning and others cover the impact of the SLOOP computational model on correctness reasoning. These topics are discussed in more detail below.

7.1.3 Reusability

The fact that the system is designed as a set of classes suggests that the procedure used to reason about the correctness of the system should take advantage of this architecture. The aim is therefore to prove the properties (informally) on a **per class** basis and then rely on those

properties to hold when the methods of such a class are invoked. This approach **simplifies the arguments** and provides **structure** to the procedure.

In order to achieve this one needs to show that if each component behaves correctly in isolation, then it behaves correctly in concert with other components [AbLa89]. In UNITY, the **restricted union rule of superposition**[1] specifies that any statement *r* may be added to the underlying program provided that *r* does not assign to the underlying variables [ChMi88]. If program composition is performed according to this rule, every property of the underlying program is a property of the transformed program [ChMi88].

In the SLOOP method it is argued that when a new class is **added** to an existing set of classes in a SLOOP program, then the behaviour of the individual classes remains correct, provided the new class does not violate any of the preconditions specified for the methods of the existing classes. Since each class **encapsulates** its own data, the class itself controls the way in which this data may be modified. The **restricted union rule of superposition** is therefore used in SLOOP when new classes are added.

When **specialization** of a class is performed, the descendant class can modify the variables inherited from its ancestors, so this is similar to the concept of **union**[2] used in UNITY programs. The union of two UNITY programs *F* and *G* contains the statements from both programs and these statements can access and modify the same variables. The union theorem given in [ChMi88] specifies how **unless** and **ensures** properties, as well as **fixed points** are preserved under union[3].

The notion of a **conditional property** was included in [ChMi88] to deal with liveness properties of the form *p* **leads-to** *q* when considering the union of *F* and *G*. In short, a conditional property of program *F* consists of a **hypothesis** and a **conclusion**, each of which is an unconditional property. The hypothesis represents the specification of a system in which *F* can be embedded. The conclusion describes the effect of embedding *F* in the system. The union of *F* and *G* is the result of embedding *F* in the system.

When a SLOOP class is being specialized, new methods are added or existing methods are overridden. Thus, the original class can be viewed as *F* and the descendant can be viewed as the union of *F* and *G*, where *G* represents the statements added by the descendant. The constraints that must be satisfied by the descendant represent the hypothesis described above. These constraints are the class and method properties (excluding liveness properties of the form *p* **leads-to** *q*) of the descendant's ancestors. By using these properties as hypotheses, conclusions can be derived that are of the form *p* **leads-to** *q* and which represent the liveness properties of the ancestors. In turn, these liveness properties can be used to derive new liveness properties for the descendant.

Since the correctness properties of a descendant should not violate any correctness properties of its ancestors, it implies that the specialization of a class should not cause any of the existing

---

[1] The concept of superposition in the UNITY context was described in more detail in Chapter 2, Section 2.5.4.

[2] A brief discussion of the union concept was presented in Chapter 2, Section 2.5.4. More details are available in [ChMi88].

[3] The union theorem given in [ChMi88] is as follows:

1. p **unless** q in F [] G = (p **unless** q in F ∧ p **unless** q in G)
2. p **ensures** q in F [] G =
   [p **ensures** q in F ∧ p **unless** q in G] ∨
   [p **ensures** q in G ∧ p **unless** q in F]
3. (**FP** of F [] G) = (**FP** of F) ∧ (**FP** of G)

classes in the system to behave incorrectly. Thus, any class that sends messages to the descendant of the original class, can still rely on the pre- and postconditions of the original methods. The preconditions of the descendant's method will not be strengthened and postconditions will not be weakened. This topic is discussed further in Section 7.3.1.4.

The correctness properties specified for a class and its methods are the responsibility of that specific class. Each class is **obliged** to guarantee that the **postconditions** of each of its properties hold, **provided** the **preconditions** are met. A property is therefore only proved once for a given class and thereafter it may be **reused** by other classes as a lemma[4]. Whenever an object invokes a method of another object, the client object has to ensure that the preconditions of the method being invoked are met.

One of the features of a SLOOP program is the fact that the statements that are executed infinitely often are **encapsulated** in parallel methods and therefore **form part of the appropriate classes** comprising the system. Exactly the same principles that apply to the sequential methods of a class therefore also apply to its parallel methods, i.e. the correctness properties of the parallel methods can be reused in the same way as those of sequential methods.

The system itself can be viewed as a composite class. The correctness properties of this class represent the required behaviour of the system[5]. Once the correctness properties of the classes that form **part** of the system have been proved (informally), the behaviour of these classes may be assumed to be guaranteed as specified in these correctness properties. When reasoning about the correctness of the **system**, these assertions can be reused as lemmas.

The correctness properties of the composite class representing the system specify the **interactions** of the objects that form part of the system. When a composite class is subclassed in order to add a new class to the system, the effect of the resulting additional or modified object interactions must be taken into account in the correctness properties of the subclass of the composite class. This applies to any composite class; not only to the one representing the system under development

A class might **reuse** the correctness properties of its ancestors, **override** them or **add** new properties. The impact of the inheritance feature of the SLOOP method is discussed in Section 7.3.1.4.

Note that design patterns can also be reused and their correctness should also be shown. However, that topic will not be dealt with in this chapter. All the issues surrounding the usage of design patterns in the SLOOP method will be covered in Chapter 9.

7.1.4 Absence of control flow

The absence of control flow in the parallel methods allows one to reason in terms of the conditions, operations and assignments of **individual** parallel statements. There is no need to consider the various **combinations of statement interleavings** in a multi-process environment, since the order in which these statements are executed is irrelevant. The **sequential methods** are invoked from the parallel methods and are viewed as **terminating functions**. Each **parallel statement** is executed **atomically**. In the correctness arguments one can rely on the fact that each statement will be executed inifinitely often.

---

[4] A lemma is defined as an "assumed or demonstrated proposition used in argument or proof" [Syke76].

[5] In the call centre example, the CC_SimulationActivation class represents the behaviour of the system.

7.1.5 Using correctness properties to derive SLOOP statements

The methods of the various classes of a system are defined once the design phase correctness properties have been specified. The behaviour described by the correctness properties yields the necessary information to derive the methods of the classes. The liveness and/or precedence properties provide the basis for deriving the parallel methods, as will be seen in Section 7.4. The infinitely often execution of the statements of the parallel methods of the various classes has to result in the desired progress being made.

The sequential methods are also derived from the correctness properties. Some of them are even referenced directly in the correctness properties themselves, as will be shown in Section 7.4. Some correctness properties are used to refine statements that have been derived from other correctness properties. This role of correctness properties will also be demonstrated in Section 7.4.

The above-mentioned features of the SLOOP method are now exemplified using the call centre example introduced in the earlier chapters.

Note: The examples below are at a detailed level. In many cases the discussions of the issues at hand are interspersed with various parts of the informal proofs. In order to highlight the points being made, the relevant parts of the text appear in boxes.

7.2 Conveying the semantics

As stated earlier, one of the purposes of specifying the correctness properties of a class and its methods is to enable the designers who wish to reuse the class to ascertain the behaviour of the class without having to study the code. This is illustrated by the discussion of the `ServiceProviderSimulator` class below. The purpose of this class is to simulate the behaviour of a service provider. The class was first mentioned in Section 5.3.2 and its analysis level properties were identified in Section 5.4.5.2. In Section 6.2.3 the `EventSimulator` class was introduced as a parent class for simulator classes. Section 6.5 contained a summary of the methods of the `EventSimulator` and `ServiceProviderSimulator` classes after the design level refinements had been performed. The full SLOOP specifications of the `EventSimulator` and `ServiceProviderSimulator` classes appear in Appendix B, Sections B.5 and B.13 respectively.

7.2.1 The static nature of a class

A designer seeking information on **the static nature** of the `ServiceProviderSimulator` class should refer to the **list of class and instance variable names of the class and its ancestors**. This list represents the **attributes** of the class and if the class is a **composite** class, some of these variables will refer to the **parts** of the composite class. The `ServiceProviderSimulator` class inherits the following variables from the `Event=Simulator` class:

```
rand
        "This variable refers to an instance of the Random class from
        the Smalltalk library.  The instance is created when the
        EventSimulator subclass is instantiated.  The instance of the
        Random class maintains a seed from which the next random number
        is generated. The random number is used to start a timer with a
        random value."
```

```
newEventRequired
        "When the value is equal to true it means that a new event is
        required.  Once the variable has been set to true, a random
        timer will be started at some point afterwards.  When the timer
        is started, newEventRequired is set to false.  It is the
        responsibility of the subclass to set this variable to true when
        a new event is required, since each subclass will have its own
        conditions for requiring a new event.  Once the timer expires,
        an event will be generated, as will be described in the comments
        section of the generatingEvent variable."
currentRandomTimeoutValue
        "This variable contains the value of the random timeout
        currently being requested.  The purpose of this variable is to
        provide a mechanism for referencing the current timeout value in
        the correctness arguments.  Note that the SLOOP statements could
        therefore have been rewritten without this variable while still
        providing the same functionality.  However, in that case it
        would not have been possible to formalise certain correctness
        properties (such as DL1-04(EventSimulator), listed in Appendix
        B, Section B.5)."
generatingEvent
        "The value is equal to true if the timer has expired and an
        event has to be generated, otherwise it is equal to false.  The
        subclass sets this variable to false at the time when the event
        is generated.  The actual event that is generated is also the
        reponsibility of the subclass, since each subclass will generate
        a different type of event."
timerOutstanding
        "This variable is set to true when a timer is started and it is
        set to false when a timeoutElement is removed from the
        timerEventQ (i.e. when an expired timer has been processed).
        The purpose of this variable is to provide a mechanism for
        reasoning about the uniqueness of outstanding timers in the
        EventSimulator class.  In this class only one timer requested by
        the EventSimulator may be outstanding at a time.  The
        timerOutstanding variable is used in the preconditions of the
        startRandomTimer:withMaximum: method as well as in the
        postconditions of the resetTimerExpired: method.  If subclasses
        need to support multiple simultaneous timers, then the
        preconditions of the startRandomTimer:withMaximum: method need
        to be weakened and the postconditions of the resetTimerExpired:
        method need to be strengthened. Since the purpose of the
        timerOutstanding variable is to facilitate correctness
        reasoning, the SLOOP statements could have been rewritten
        without this variable while still providing the same
        functionality."
timerId
        "This variable contains the identifier of the timer currently
        being requested.
```

From the above description the designer learns that the `EventSimulator` is a composite class, since the `Random` class forms part of it. The purpose of each variable is also gleaned from the comment following each variable name. As pointed out in the relevant comments, some variables are introduced solely for the purpose of reasoning about the correctness of the class.

For example, the statements

```
  currentRandomTimeoutValue :=
      (self nextRandomNumber: maximumValue)
[] aTimerServices start: self id: timerId for:
      currentRandomTimeoutValue
```

could have been replaced by

```
[] aTimerServices start: self id: timerId for:
        (self nextRandomNumber: maximumValue)
```

thereby eliminating the `currentRandomTimeoutValue` variable from the SLOOP statements in the `startRandomTimer:withMaximum:` method. However, a similar replacement would then have been required in the correctness properties referencing this variable. Unfortunately this is not possible, since the `(self nextRandomNumber: maximumValue)` expression has side-effects and may therefore not be used in correctness properties. The rationale for this restriction is the fact that correctness properties are at a **meta-level,** i.e. they describe the behaviour of the system and should **never modify** the system state.

The usage of the `timerOutstanding` variable is shown in Appendix B, Section B.5.

The `ServiceProviderSimulator` subclass adds a number of variables to the above list:

```
instance variable names
serviceRequest
        "This variable refers to the service request currently being
        serviced by the service provider simulator.  Note that the
        reference to the ServiceRequest instance is passed to the
        simulator as a parameter, i.e. the ServiceRequest instance is
        not created by the ServiceProviderSimulator instance and
        therefore does not form part of it."
serviceProviderCategory
        "This variable contains the name of the service provider
        category to which the service provider simulator belongs."
categoriesServed
        "This is an ordered collection containing the names of the
        service request categories serviced by this service provider.
        The purpose of this array is to facilitate a round robin
        servicing scheme of these categories. That prevents starvation
        of a specific service category."
nrOfCategoriesServed
        "This variable contains the number of service request categories
        serviced by this service provider. It is used in the calculation
        when the categoryIndex is updated."
categoryIndex
        "This variable is used as index into the categoriesServed
        collection. It is used to determine the next service request
        category to be serviced by this service provider. It is
        incremented modulo nrOfCategoriesServed. Its values range from 0
        to nrOfCategoriesServed - 1"
```

The purpose of the `categoriesServed`, `nrOfCategoriesServed` and `categoryIndex` instance variables is to prevent a `ServiceProviderSimulator` instance from ignoring one of the service queues that it should be servicing for ever. The `ServiceProviderSimulator` class is a composite class (it inherits the `Random` class from its `EventSimulator` parent class). The `categoriesServed` instance variable contains a reference to another part of the composite class, viz. `OrderedCollection`.

7.2.2 The dynamic nature of a class

To find out about the **dynamic behaviour** of a service provider simulator, one would have to become acquainted with the various **properties** of a **class and its ancestors**. The properties listed below are the **class** properties of the parent of the `ServiceProviderSimulator` class, namely the `EventSimulator` class. Subsequently the **method** properties of the `EventSimulator` class will be examined, followed by the inspection of the `ServiceProviderSimulator` **class and method** properties. Note that the **analysis level**

properties are only specified **informally**. Where **possible,** the **design level properties** are specified **formally**. The rationale for only giving informal specifications of some of the design level properties is given later.

```
class properties
        "Liveness"
        "When a simulation event is required, a simulation timer is eventually started."
                                                    "AL2-01 (EventSimulator)"

        "Liveness"
        "If a simulator timer expires, the simulator eventually has to generate an event."
                                                    "AL2-02 (EventSimulator)"

        "Clean behaviour"
        <∀ anObject ::
                    invariant anObject class ~~ EventSimulator
        >                                           "DS2-01 (EventSimulator)"
        "The EventSimulator class is an abstract class and should not be instantiated."

        "Clean behaviour"
        invariant    rand notNil ∧ rand class = Random
                                                    "DS2-02 (EventSimulator)"
        "Once rand has been initialized to refer to an instance of the Random class, it is never
        set to nil while the instance of the EventSimulator subclass exists."

        "Clean behaviour"
        "The currentRandomTimeout value is always within the range specified by the
        precondition of the start:id:for: method of the TimerServices class."
                                                    "DS2-03 (EventSimulator)"

        "Global invariant"
        "All outstanding timers requested by an EventSimulator subclass instance are identified
        uniquely with respect to the requestor."
                                                    "DS3-01 (EventSimulator)"
```

The properties are organised first according to the development phase from which they originated and next according to the property type. The analysis level properties are presented first, since they provide the designer with the gist of the functionality of the class. Properties *AL2-01* and *AL2-02* convey to the designer that a timer will be started if a new event is required and when that timer expires, an event will be generated.

The last four properties are design level safety properties. By inspecting them, the designer is able to learn about the design of the `EventSimulator` class. The first property reveals that the `EventSimulator` class is an abstract class. The second property guarantees clean behaviour as far as the composite and its parts are concerned: it ensures that subclass instances of the `EventSimulator` class will never send messages to a non-existing instance of the `Random` class.

The next clean behaviour property specifies that the method to start a new timer will always be invoked with a requested timer value that is within the range as specified in the precondition of the method being invoked. The **reason why this property is not specified formally at this point** is because it refers to the requirements of another class. The **reference to this other class** is passed as an **argument** to the `p_simulate:timeoutEventsIn:` method of the `EventSimulator` class and the formal specification of this property is therefore given within that method. The interested reader is referred to Appendix B, Section B.5 for details.

The fourth property is a global invariant which ensures that the notification of the expiry of an `EventSimulator` timer can be correlated with the correct timer request. It specifies that each outstanding timer requested by a subclass instance of the `EventSimulator` class is identified uniquely with respect to its requestor. This property is specified **informally only** in this example.

There are numerous ways in which this property can be specified formally. One possibility is formulate it in terms of the currently outstanding timers and the identifiers associated with them. That implies that several additional instance variables would have to be defined and maintained. It is **debatable whether the additional complexity is justified** in the case of the `EventSimulator` class, where only one timer may be outstanding at a time. Should a subclass require multiple simultaneous outstanding timers, the modifications required to support those timers might also, as a side-effect, yield the necessary additional variables to facilitate a formal version of the above property. For these reasons, it was decided that the informal version of property *DS3-01* would suffice for the `EventSimulator` class.

As stated earlier, the identification of correctness properties during a specific software development phase is **not** an **ad hoc** process. The checklist as presented in Section 5.2.4 is used to aid the software designer in following a **structured** and **systematic** approach when first recording these properties. When they are subsequently inspected in order to evaluate the class for potential reuse in other systems, different designers may prefer different groupings of the properties. One possibility is to group the properties according to functionality. For example, for the `EventSimulator` class all properties related to starting a timer could be grouped together and similarly all properties related to the expiry of a timer could be grouped together. Combining the properties in a different way is especially useful during the familiarisation process.

However, the SLOOP method favours the **recording** of the properties based on the checklist as discussed above. One of the reasons for this preference is that it addresses one of the problems associated with the conjunctive nature of logic-based methods, viz. the **completeness** of the specification [JiZh96]. Although one cannot guarantee the completeness of such a specification [Fran92], at least one is guided to consider each property type and to reflect whether or not there are any useful properties that can be specified for each type.

In this example the next step is to inspect the properties of the **individual methods** of the **superclass** (i.e. the `EventSimulator` class). That is followed by the inspection of the **class** properties of the **subclass** (i.e. the `ServiceProviderSimulator` class), after which the properties of the `ServiceProviderSimulator` **methods** are examined. However, the order in which these steps are performed is not prescribed by the SLOOP method.

The following methods are defined for the `EventSimulator` class:

```
initialize
nextRandomNumber:
startRandomTimer:withMaximum:
timerExpired:
resetTimerExpired:
p_simulate:timeoutEventsIn:
```

A great deal can be learnt from the correctness properties of these methods. For brevity, this information is summarised here. The interested reader is referred to Appendix B, Section B.5 for the SLOOP specification of the correctness properties of the above methods.

The `initialize` method is executed when an `EventSimulator` subclass is created. It ensures that all the instance variables of the `EventSimulator` class will have initial values. The descendants may perform some additional initialisation, but the total correctness property of

the `initialize` method guarantees that at least the `EventSimulator` variables will not be uninitialised.

The `nextRandomNumber:` method always returns a number ranging from 1 to `maximumValue`, where `maximumValue` is passed to the method as an argument. The `startRandomTimer: withMaximum:` method guarantees that a timer is started with a value within the range from 1 to the maximum value received as one of the arguments of the method. This method invokes the `nextRandomNumber:` method in order to generate the timer value.

The `timerExpired:` method returns true if a timer requested by the `EventSimulator` subclass instance has expired and false if not. This is determined by the presence or absence of the relevant `timeoutElement` in the `timerEventQ`. The `resetTimerExpired:` method removes a `timeoutElement` from the `timerEventQ`.

Finally, the liveness properties specified for the `EventSimulator` class are realised via its parallel method. The `p_simulate:timeoutEventsIn:` method ensures that a timer is started if `newEventRequired` is true and it guarantees that `generatingEvent` will be set to true once the timer has expired.

It is clear from these properties that the `EventSimulator` class has nothing to do with the actual event that is generated. It also does not determine when a new event is required. These are functions of the subclasses. In order to find out **when a new is event is required** and **what type of event is generated** in the case of a `ServiceProviderSimulator` class, the properties of that subclass are examined next.

class properties

```
<∀ categoryIndex where categoryIndex ≥ 0 ∧
categoryIndex ≤ nrCategoriesServed - 1 ::
invariant serviceRequest notNil ⇒ ¬self canAcceptNextSR:
      (categoriesServed at: (categoryIndex + 1))
>                          "AS3-01 (ServiceProviderSimulator)"
```

"A service provider simulator services a single service request at a time."

```
"If a service request is currently assigned to the simulator, no
other service request from any of the categories being served by
this simulator will be served by the latter."

serviceRequest isNil ∧ ¬newEventRequired unless
serviceRequest notNil ∧ newEventRequired
                          "AS4-01 (ServiceProviderSimulator)"
```

"When a new service request is assigned to the service provider simulator then a new service provider simulator event is required."

"Note: The parent class, viz. EventSimulator, contains a parallel method which monitors the value of newEventRequired. If it detects that newEventRequired is true, it starts a timer and sets newEventRequired to false."

```
serviceRequest notNil ∧ ¬newEventRequired unless
serviceRequest isNil ∧ ¬newEventRequired
                          "AS4-02 (ServiceProviderSimulator)"
```

"If a service request is assigned to the service provider simulator and newEventRequired is false, then newEventRequired remains false for at least as long as the service request is still assigned to the service provider simulator."

```
"This has the effect that this simulator will not start another
timer before the servicing of the current service request has
been completed."

generatingEvent ∧ serviceRequest notNil ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil ∧
      ¬generatingEvent   "AP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the service provider simulator terminates the connection currently associated with it and becomes available to service a new service request."

"Note: The parent class, viz. EventSimulator, contains a parallel method which sets generatingEvent to true when a timer has expired."

```
<∀ aServiceRequest where serviceRequest = aServiceRequest ::
 serviceRequest = aServiceRequest ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil
>                          "AP1-02 (ServiceProviderSimulator)"
```

"A service request remains assigned to a service provider simulator until the latter completes the service and terminates the connection."

```
"Clean behaviour"
invariant categoryIndex ≥ 0 ∧
      categoryIndex < nrOfCategoriesServed
                           "DS2-01 (ServiceProviderSimulator)"
```

"The categoryIndex is always greater than or equal to zero and less than nrOfCategoriesServed."

```
"Clean behaviour"
invariant    categoriesServed notNil ∧
             categoriesServed class = OrderedCollection
                           "DS2-02 (ServiceProviderSimulator)"
```

"Once categoriesServed has been initialized to refer to an instance of the OrderedCollection class, it is never set to nil while the ServiceProviderSimulator instance exists."

```
<∀ categoryIndex where 0 ≤ categoryIndex ∧
categoryIndex < nrOfCategoriesServed ::
¬(self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1)))
leads-to
      self canAcceptNextSR:
      (categoriesServed at: (categoryIndex + 1))
>                          "DL2-01 (ServiceProviderSimulator)"
```

"For any service category serviced by the service provider simulator, the service provider simulator will eventually be able to service a request from that service category."

Property *AS4-01* (the first **unless** property) reveals how `newEventRequired` is set to true, viz. when a new service request is assigned to the simulator. The statements of the parent class ensure that a timer is started if `newEventRequired` is true. When the timer is started, `newEventRequired` is set to false.

The `newEventRequired` variable then remains false until the service has been completed. The latter happens once the timer has expired and the service request is deallocated from the simulator. This is evident from properties *AS4-02*, *AP1-01* and *AP1-02*. Properties *AS3-01* and *AP1-02* guarantee that only one service request is serviced at a time. Property *AP1-01* provides information regarding the nature of the event that is generated. Property *DS2-01* specifies the allowed values of `categoryIndex`, property *DS2-02* guarantees that the `categoriesServed` variable will not be nil and property *DL2-01* ensures that the service provider simulator will service each service category supported by it.

The properties of the individual methods are now inspected in order to obtain more information about the ServiceProviderSimulator class. These methods are as follows:

```
startSimulation:using:
moreInit:using:
registerServiceProvider:using:
serviceProviderCategory
serviceRequest
canAcceptNextSR:
processServiceRequest:
p_generateEvent
p_updateCategoryIndex:
```

Details of the correctness properties of the methods listed above are provided in Appendix B, Section B.13. For the purposes of this discussion it suffices to point out that the first three methods are used when a `ServiceProviderSimulator` instance is created. These methods ensure that the `serviceRequest` variable is set to nil, that a service provider category is assigned to the simulator and that the simulator is registered with the relevant service categories during instance creation. These methods are also responsible for creating the `categoriesServed` ordered collection and for recording all the service categories supported by this `ServiceProviderSimulator` instance in that collection.

The `serviceProviderCategory` and `serviceRequest` methods are accessing methods that return the category of the service provider simulator and the current service request being serviced respectively.

The `processServiceRequest:` method is invoked by the client of the `ServiceProviderSimulator` instance in order to assign a new service request to the simulator. The client has to ensure that the precondition of the `processServiceRequest:` method holds when it invokes it. The `canAcceptNextSR:` method forms part of the precondition. It returns true if the `serviceRequest` variable is equal to nil and the message argument matches the next service category to be serviced by this simulator. It returns false otherwise. By using the `canAcceptNextSR:` method in the precondition of the `processServiceRequest:` method, it can be guaranteed that a new service request is only accepted if no other service request is currently being serviced by this simulator. It also ensures that the service categories are serviced in a round robin fashion.

The postconditions of the total correctness property of the `processServiceRequest:` method specify that `newEventRequired` will be true and `serviceRequest` will not be nil when the method has completed its execution. Thus it ensures that `newEventRequired` is set to true when a new service request is assigned to the simulator. In turn, the parallel method of the `EventSimulator` class ensures that a timer is started if `newEventRequired` is true. Once the timer expires, `generatingEvent` is set to true (also via the parallel method of the parent class). The precedence property of the `p_generateEvent` parallel method then guarantees that the relevant event is generated if `generatingEvent` is true.

The `categoryIndex` instance variable is updated in the `processServiceRequest:` and `p_updateCategoryIndex:` methods. As is evident from the correctness properties of these methods, the value is updated in a way which ensures that the relevant service queues can be serviced in a round robin fashion by this simulator. The `processServiceRequest:` method ensures that the `categoryIndex` is updated whenever a new service request is accepted from a service queue associated with the service category indicated by the current value of `categoryIndex`. The parallel method, `p_updateCategoryIndex:`, checks the service queue associated with the service category indicated by the current value of `categoryIndex`. If that service queue is empty, the `categoryIndex` is also updated.

In summary, information about the **static** nature of a class is obtained via the class and instance **variables** of the specified class and its ancestors. The software designer needs to inspect the **correctness properties** of the class and its methods, both of the class itself and of its ancestors, in order to learn about the **dynamic behaviour** of the class.

The purpose of the discussion in this section was merely to demonstrate how **information about the behaviour** of the class could be obtained from the correctness properties specified for the class and its methods. No correctness arguments were given. In the sections that follow, it will be demonstrated how one can use informal correctness arguments to show how the correctness properties of the relevant individual methods ensure that various correctness properties of the class are satisfied. That description forms part of a discussion of the impact of various features of the SLOOP method on correctness reasoning. It is the topic of the next section.

7.3 The impact of various SLOOP features on correctness reasoning

The purpose of this section is to illustrate how various features of the SLOOP method result in a **gain in simplicity** as far as correctness reasoning is concerned. Examples of informal arguments to reason about safety, liveness and precedence properties are given next. For each property **type** listed in Chapter 5, the correctness arguments for at least one property are presented. The software designer would use such arguments during the software development process to verify informally that already stated correctness properties indeed hold.

Note that the SLOOP method **does not mandate** that these correctness arguments form part of the official documentation of a project. One could therefore choose to adopt this style of thinking without recording these informal proofs. However, by documenting the correctness arguments they can be checked by others and they are then also available to those who might want to reuse the classes to which the correctness arguments apply.

Apart from illustrating how one can reason about different types of correctness properties, the specific properties in the examples below are also chosen to highlight additional issues. For example, in Section 7.3.1.2 it is shown how correctness reasoning is simplified due to the fact that **program location counters** can be **ignored** when the **atomic** units of execution (the parallel statements) are considered. In Sections 7.3.2.3 and 7.3.3.2 it is illustrated how the distinctive properties of the **leads-to** and **ensures** relations respectively are utilised in correctness arguments.

In Chapter 5 the desired analysis level properties of the call centre system were stated. These properties describe the interactions of the objects that comprise the system and are captured within the `CC_SimulationActivation` class and its superclass, `CC_Activation`. The system properties listed in Chapter 5 were written in terms of the analysis phase artifacts, i.e. prior to the design level refinements. For example, during the analysis phase the

`ConnectionContainer` class was defined as the container class of the `Connection` instances. As a result of the design phase refinements, it was found that the `ConnectionContainer` class was superfluous. Instead, the `Array` class from the Smalltalk library sufficed. The correctness properties that include the design level refinements refer to this `Array` instance as `userConnections`.

The correctness properties as discussed in the remainder of this chapter include the design level refinements. Note that properties that emanated from the analysis phase retain their identifiers as assigned to them during that phase. Thus, even though design phase refinements have been added, their identifiers do not change, since their origin (i.e. the analysis phase) remains the same.

The correctness properties of a class may be overridden in a descendant, as will be demonstrated in Section 7.3.1.4. In that case the property in the subclass has the same identifier as the one in its ancestor, but the identifier is followed by the name of the **ancestor** in brackets. When referring to such a property **from within another class**, the words "in *class-name*" have to be added, where *class-name* is the name of the class which overrides the property.

7.3.1 Safety properties

In this subsection the safety properties relating to clean behaviour, global invariants and the unless relation are considered. Each subsection sheds light on something specific:

- Section 7.3.1.1: the advantages of using **macros** with respect to correctness reasoning;
- Section 7.3.1.2: the influence of the **SLOOP computational model** on correctness reasoning and the impact of the object-oriented features such as **data encapsulation** and **reusability** on correctness arguments;
- Section 7.3.1.3: the **allocation of responsibilities** regarding the preconditions during method invocations; and
- Section 7.3.1.4: the **effect of inheritance** on correctness arguments.

The generic approach towards proving a correctness property informally can be described as follows: First of all the correctness property is specified formally. Then the strategy to be followed in order to prove informally that the property holds is determined. Thereafter the correctness arguments are given. The latter can be presented from first principles (i.e. by inspecting the statements of the SLOOP program), or correctness properties that have already been proven, can be reused if applicable. The above procedures are described in detail in Section 7.3.1.2.

7.3.1.1 Using the correctness arguments of a clean behaviour property to illustrate how the use of macros in SLOOP programs can simplify correctness reasoning

The purpose of this section is to illustrate the **advantages of using macros** in SLOOP programs with respect to correctness reasoning. This is achieved by discussing the correctness arguments of property *AS2-01(CC_Activation)*, **an analysis level clean behaviour safety property**. It is one of the safety properties of the `CC_Activation` class and specifies the following:

invariant `userConnections capacity = maxConn ∧ maxConn > 0`
"AS2-01 (CC_Activation)"

*"The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer."*

This property is used to describe the capacity restrictions of the call centre. The purpose of this property was discussed in detail in Chapter 5, Section 5.4.1.2. In order to show that the above property always holds for the `CC_Activation` class and its subclasses, the contents of this property first needs to be examined more closely.

First of all, it sends the `capacity` message to `userConnections`, the `Array` instance. One therefore needs to look at the characteristics of the `Array` class and its methods. The `capacity` method returns the number of indexed instance variables of the `Array` instance. The indexed variables are used to hold the elements of the `Array` instance. When an `Array` instance is created, all its indexed instance variables are created. The number of indexed instance variables remains fixed throughout the existence of the `Array` instance. It is equal to the value of the parameter that is passed to the `Array` class when the `new:` method is invoked.

In order to show **informally** that the capacity of `userConnections` is equal to `maxConn` and does not change, it needs to be shown that `userConnections` is created as an `Array` instance of this capacity. This is done by inspecting the SLOOP program statements as given in Appendix B. The `userConnections Array` instance is created in the `initialize` method of the `CC_Activation` class[6] via the statement below:

```
userConnections := SmalltalkLibPkg:::Array new: maxConn
```

It is the only statement that assigns a value to the `userConnections` variable in the `CallCentreSimulation` program. Thus, this statement ensures that the `capacity` of the `userConnections` object is equal to `maxConn` and it therefore means that `userConnections` can hold exactly `maxConn` number of elements.

The second part of property *AS2-01* states that `maxConn > 0` is an invariant. This part of the example highlights the **advantages** of using **macros** in a SLOOP program. There are a number of references to `maxConn` in the `CC_Activation` class, but `maxConn` is not an attribute of this class, as is evident from the list of instance variables of this class in Appendix B, Section B2. Instead, all of these references are expanded to `config maximumConnections`, the message expression which obtains the value of *maximumConnections* from an instance of the `Configuration` class. One of the invariants of the `Configuration` class[7] is the following:

```
<∀ ( t, u, v, w) where
t > 0 ∧ u > 0 ∧ v > 0 ∧ w > 0 ::
invariant
        maximumConnections = t ∧
        maximumServiceCategories = u ∧
        maximumServiceProviders = v ∧
        maximumAllowableTimeout = w
>                                          "DS2-01(Configuration)"
```

Thus, the value of `maximumConnections` is always greater than zero and it also does not change. Since property *DS2-01* is an invariant of the `Configuration` class, it means that once the class is instantiated, this property holds for that class. All other classes that send the `maximumConnections` message to the `Configuration` instance may therefore assume that it will return a value that is greater than 0, since it is guaranteed by the `Configuration` class.

---

[6] The `initialize` method of the `CC_Activation` class is presented in Section B.2 of Appendix B.

[7] The Configuration class is specified in Appendix B, Section B.4. The rationale for having such a class was given in Chapter 6, Section 6.3.1.

The **advantage of using a macro** rather than an instance variable in the above case is evident from the fact that **correctness property *DS2-01* of the `Configuration` class may be reused**. Since a *macro-variable* may not appear on the left-hand side of any SLOOP statement, it can never contain any value other than the one to which its associated *macro-expression* evaluates.

One alternative to the above approach is for each client class to define an instance variable representing the maximum number of connections. It is then the responsibility of each class to ensure that this variable always contains the correct value (which implies that a correctness property stating this has to be defined for each client class and the correctness of this property also needs to be shown).

This example gives an **indication** of the **nature** of the correctness arguments that are used in the SLOOP method. It also gives an **indication** of the **level of rigour** that is present in these arguments. The next section provides more detail regarding the steps that are followed during correctness reasoning in the SLOOP method.

7.3.1.2 Demonstrating how the computational model and object-oriented features such as data encapsulation and reusability influence the approach followed during correctness reasoning

Another clean behaviour property of the call centre system, viz. property *AS2-04* defined in Chapter 5, Section 5.4.1.2, is now used to illustrate the general approach followed during informal correctness reasoning in the SLOOP method.

The purpose of this section is twofold, viz.

- Firstly, it demonstrates how the **computational model** influences the correctness arguments. It is shown how **induction** is used when proving **invariance** properties informally.
- Secondly, it illustrates how object-oriented features such as **data encapsulation** and **reusability** are taken advantage of in the correctness arguments.

Informally, property *AS2-04(CC_Activation)* is defined as follows:
If a connection is terminated, it implies that its associated service request is not present in the input queue.

The approach has three major steps:

- The first step is to specify the property formally. (It is difficult to reason about something that can be interpreted in different ways.) Thus, the property has to have an unambiguous meaning.
- The strategy required to arrive at the property as conclusion, is determined.
- Arguments are presented to support the various claims as outlined in the strategy.

Specifying the correctness property formally

In order to specify property *AS2-04* more formally, one needs to consider how a terminated connection should be represented. In Section 6.3.2 it was explained that a design decision was made to create the maximum number of `Connection` instances upon startup and to use the state of an instance to determine whether it represented an unassigned connection or not.

The rationale for making this decision was given in that section. Briefly, it ensures that the parallel statements of all `Connection` instances are always present to be selected for execution. As stated in [ChMi88], it would be difficult to define a fair execution rule for statements if the set of program statements changed dynamically. If `Connection` instances could be created and destroyed dynamically after initialization, then it would imply that their associated methods would only be present during the existence of the instances.

When the analysis level property *AS2-04 (CC_Activation)* was formulated in Chapter 5, the aim was to avoid any design level detail. As a result, there were no references to the `state` of a `Connection` instance or the presence or absence of `Connection` instances in the `userConnections` collection in order to indicate the availability of a connection. The terminology used in the specification of the property was at a very high level of abstraction.

Once the design decision had been made to reflect the availability of the `Connection` instance via its `state`, the property was rewritten as follows:
If a connection is idle, it implies that its associated service request is not present in the input queue.
More formally:

```
<∀ aConnection where userConnections include: aConnection ::
      invariant    aConnection isIdle  ⇒
                   ¬(inputQ includes: (aConnection serviceRequest))
>                                                  "AS2-04 (CC_Activation)"
```

Below correctness arguments are given to show that property *AS2-04(CC_Activation)* is indeed invariant.

The strategy to be followed for the correctness arguments

The correctness arguments in methods based on the conventional computational model take the location counters of the processes involved in the computation into account. For example, in [MaPn81a] the proof principles presented for establishing correctness properties clearly take cognisance of the location counters. In the SLOOP correctness arguments that are employed at the design level, there is no notion of concurrent processes and therefore they **do not contain references to location counters** of different processes. Instead, the correctness arguments show that there **exist** statements in the program that will ensure that the specified **progress** properties will hold and also that **no statements exist** that will violate the **safety** properties.

The following statements of the `CallCentreSimulation` program are therefore considered:

- The **sequential statements** in the *activation-section* of the program (in the order of their appearance).
- All the **parallel statements** that are activated directly or indirectly via the *activation-section* of the program (in any order).

The **sequential methods invoked by the above statements** are also taken into account.

The principle of **induction** [MaPn81b] is used when proving **invariants** in the SLOOP method. Considering the SLOOP computational model, this means that one has to show that the property holds initially, as well as after the execution of any parallel statement of the program.

The correctness arguments for property *AS2-04(CC_Activation)* are presented in two parts. Part A shows that the property holds initially. Part B contains the arguments to prove informally that the property also holds after the execution of each parallel statement of the program.

The arguments in part B are in support of two claims, viz.:

B1) The connection state changes to **not** 'IDLE' whenever the service request is added to the input queue.
B2) The state does not change to 'IDLE' while the service request is still present in the input queue.

Thus, while a service request is present in the input queue, the associated connection is not 'IDLE'. This implies that if a connection is 'IDLE', it cannot be present in the input queue, which is what invariant *AS2-04(CC_Activation)* specifies.

The correctness arguments

The correctness arguments for property *AS2-04* are now presented. In part A it is shown that the property holds immediately after initialization. In part B1 arguments are presented to prove informally that any statement that adds a service request to the input queue also changes the state of the connection to not 'IDLE' at that point. In part B2 it is shown that no statements exist that will change the state of the connection to 'IDLE' while the associated service request is still present in the input queue.

Part A:

In order to prove informally that property AS2-04(CC_Activation) holds **immediately after initialization**, the **sequential** statements in the *activation-section* are scrutinised. In the `CallCentreSimulation` program there is only one statement in the sequential part as can be seen in this SLOOP program excerpt:

```
program CallCentreSimulation
      sequential
      aCCSimulationActivation :=
            CC_ActivationPkg:::CC_SimulationActivation setup
      end-sequential
      parallel
      aCCSimulationActivation p_activate
      end-parallel
      "Packages"
      ...
end-program
```

The sequential statement sends the `setup` message to the `CC_SimulationActivation` class, which results in a new instance being created and initialized. The sequential methods that are invoked as part of the initialization are listed in Sections B.2 and B.3 of Appendix B. Inspection of these methods reveals that the `inputQ` is created via the following statement in the `initialize` method of the `CC_Activation` class:

```
inputQ := SmalltalkLibPkg:::OrderedCollection new: maxConn
```

However, there are no statements in the sequential methods that are invoked as part of the initialization that add service requests to the `inputQ`. Immediately after initialization the `inputQ` therefore contains no service requests.

Each `Connection` instance is created via the following statement in the `initConnection:` method of the `CC_Activation` class:

```
^CC_CorePkg:::Connection setup: index
```

As can be seen from the statements of the `initialize:` method[8] which is invoked as a result of the execution of the above statement, each `Connection` instance is in the 'IDLE' state after initialization.

Thus, initially each `Connection` instance is in the 'IDLE' state and the `inputQ` contains no service requests, which means that property *AS2-04(CC_Activation)* is satisfied.

Part B:

The next step is to prove informally that property *AS2-04(CC_Activation)* is preserved by the **parallel** statements of the program. The **amount of effort required is reduced** by the fact that only those parallel statements that are **relevant** to this property need to be considered. This is because each parallel statement forms part of a method that belongs to a class, and as a result of **data encapsulation**, the effect of the execution of a parallel statement is limited to the state of the objects that are known within that class instance. As a result only a limited number of statements need to be examined at a time. The steps described below therefore take advantage of the **structuring capabilities** inherent in an object-oriented method.

Part B1:

In order to show that the connection state changes to **not** 'IDLE' whenever the service request is added to the input queue, all the statements that add a service request to `inputQ` need to be found. The *activation-section* contains only one parallel statement, viz.

```
aCCSimulationActivation p_activate
```

In turn, the `p_activate` method[9] of the `CC_Activation` class contains several parallel statements:

```
"p_activate method of the CC_Activation class"
parallel
        self p_executeCPAgent
        "The parallel methods of the commsAgent are not invoked
        directly, but rather via the p_executeCPAgent method of the
        CC_Activation class."

        [] timer p_runTimer: timerEventQ
        "Activate the parallel methods of the timer object.  The timer
        parallel statements have the following functionality: Whenever a
        timeout occurs, the TimeoutElement instance representing the
        timeout is added to the end of the timerEventQ, which indicates
        to the requestor that the specified timer has expired."

        [] self p_categoriseAndAllocate
        "The parallel methods of the scAllocator object are invoked via
        the p_categoriseAndAllocate method of the CC_Activation class.
        The scAllocator parallel statements have the following
        functionality: Once a service request has been categorised, it
        is removed from the inputQ and appended to the appropriate
        serviceQ."
```

[8] The Connection class is specified in Appendix B, Section B.7.
[9] The p_activate method is specified in Appendix B, Section B.2.

```
        [] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
                    p_execute
          >
        "Activate the parallel methods of the ServiceCategory instances.
        Their parallel statements have the following functionality: For
        each service category the associated service queue and set of
        service provider agents are monitored.  If the service queue is
        not empty and a service provider agent in the spSubset
        associated with the service category is available to process a
        new service request, the first element of the service queue is
        removed and assigned to a service provider agent."

        [] < [] i where 1≤i≤maxConn :: self p_executeConnection:
              (userConnections at: i)
          >
        "The p_executeConnection method of the CC_Activation class is
        executed for each Connection instance in order to invoke the
        parallel methods of the latter.  The parallel statements of the
        Connection instances have the following functionality: When a
        connection has entered the 'TERMINATING' state, the
        communication provider agent is requested to terminate the
        connection.  Once all the procedures have been completed to
        terminate the connection, the connection and its associated
        service request are reset to their initial states."

        [] < [] k where 1≤k≤maxSP :: self p_executeSPAgent:
              (spAgentContainer at: k)
          >
        "The parallel methods of the service provider agents are not
        invoked directly, but rather by executing the p_executeSPAgent
        method of the CC_Activation class for each of the service
        provider agents."
end-parallel
```

These statements can be inspected in any order. For brevity, only the statement containing the `p_executeCPAgent` message is discussed here, since it is the one leading to the statement that assigns service requests to the `inputQ`. The `p_executeCPAgent` method is the responsibility of the subclass, which is the `CC_SimulationActivation` class[10] in this case. In the latter, this method contains the following statements:

```
"p_executeCPAgent method of the CC_SimulationActivation class"
parallel
        commsAgent p_simulate: timer timeoutEventsIn: timerEventQ
      [] commsAgent p_generateEvent: userConnections target: inputQ
end-parallel
```

The second parallel statement in this method invokes the `p_generateEvent:target:` method of the `CommsProviderSimulator` class[11], which contains the only statement in the `CallCentreSimulation` program that adds a service request to the `inputQ`:

```
"p_generateEvent:target: method of the CommsProviderSimulator class"
parallel
         inputQ addLast: (idleConnection serviceRequest) \+
         idleConnection assign
            if generatingEvent and: [idleConnection notNil] ~
         Transcript show: 'All connections busy'
            if generatingEvent and: [idleConnection isNil]
```

[10] The CC_SimulationActivation class is presented in Appendix B, Section B3.
[11] The CommsProviderSimulator class is specified in Appendix B, Section B.6.

```
        || newEventRequired := true \+
           generatingEvent := false
              if generatingEvent
end-parallel
```

The `p_generateEvent:target:` method of the `CommsProviderSimulator` class makes use of the `idleConnection` macro, which is defined in the `p_generateEvent:target:` method as:

```
idleConnection ≡ self getIdleConnection: userConnections
```

In turn, the `getIdleConnection` sequential method of the `CommsProviderSimulator` class is defined as:

```
^userConnections detect: [:each | each isIdle] ifNone: [nil]
```

Finally, the `isIdle` method of the `Connection` class[12] is defined as:

```
^state = 'IDLE'
```

Thus, the `p_generateEvent:target:` method has the following functionality: If an event has to be generated, an idle connection is searched for by checking whether there is any `Connection` instance in the `userConnections` array that is in the 'IDLE' state. If one is found, the associated service request is added to the `inputQ` and the state of the connection is changed to indicate that it is connected to a service user. The `assign` method of the `Connection` class contains the following statement:

```
state := 'CONNECTED'
      if state = 'IDLE'
```

If an idle connection cannot be found, a status message is generated.

From the statement in the `p_generateEvent:target:` method it is clear that a service request may only be added to the `inputQ` if the associated connection is in the 'IDLE' state. In a **single** multiple-assignment statement the service request is added to the `inputQ` and the state of the connection is changed to 'CONNECTED'. **Note that it is crucial that the insertion of the service request into the `inputQ` and the modification of the state of the connection to 'CONNECTED' should occur simultaneously, i.e. as part of the same atomic statement. This is essential for the correctness arguments of the above property, since it has to be shown that the service request cannot be added to the input queue while the connection remains in the 'IDLE' state.** This concludes Part B1 of the correctness arguments.

Note that in the above discussion it was shown how to locate a specific statement (i.e. one that adds an element to the `inputQ`) manually. In practice, any editing or browsing tool can be used to search for a statement which invokes a specific method, which makes it very simple to isolate the statements that need to be considered.

Part B2:

For Part B2 it has to be shown that the connection does not enter the 'IDLE' state while the service request is in the input queue.

The main arguments comprising Part B are as follows:
B2.1) The only transition to the 'IDLE' state is from the 'TERMINATING' state.
B2.2) At this level of refinement the method that results in a transition to the 'TERMINATING' state is not invoked while the service request is in the `inputQ` (i.e. the connection is not aborted or rejected). Only the service provider simulators invoke this method.

[12] The Connection class is specified in Appendix B, Section B.7.

B2.3) A service provider simulator only invokes this method if the service provider simulator has a service request assigned to it.
B2.4) Once a service request has been assigned to a service provider simulator the service request is no longer present in the `inputQ` and after a service request has been removed from the input queue, it remains outside the queue until the connection has been terminated and has reached the 'IDLE' state.

Part B2.1:

In this part it needs to be shown that the only transition to the 'IDLE' state of a connection is from the 'TERMINATING' state. The informal correctness arguments for B2.1 take advantage of some of the object-oriented characteristics of the SLOOP method, viz. its **structuring** and **data encapsulation** features. Instead of inspecting each statement of the SLOOP program, one first checks whether it is not possible to restrict the number of statements that need to be examined.

This is done by inspecting the `Connection` class in order to find out whether it provides any methods enabling clients to set its `state` instance variable to 'IDLE'. The idea is that if no method is provided to clients to set the state to 'IDLE', **the number of statements that need to be inspected is reduced** from all the statements in the program to only those of the `Connection` class.

The first thing that is discovered is the fact that there is no `state:` method[13] which allows a client to set the `state` to any value. Instead, the `Connection` class controls the values of the `state` instance variable by only providing an `assign` and a `terminate:` method to modify the value of `state`. The `assign` method sets the value to 'CONNECTED' and the `terminate:` method sets it to 'TERMINATING'.

Although the `initialize:` and `p_doWrapUp` methods set the value of `state` to 'IDLE', these are private and parallel methods respectively. A private method is not accessible to clients, while a parallel method cannot be invoked from within any sequential method (this rule was specified in Chapter 4, Section 4.2.3). The `p_doWrapUp` method is only invoked from within the `CC_Activation` class in order to **activate** the parallel statements of that method. Thus, the `Connection` instance itself is the only object that can set `state` to 'IDLE'. The only statement that sets `state` to 'IDLE' after instance creation and initialization, is the following one in the `p_doWrapUp` method:

```
"p_doWrapUp method of the Connection class"
parallel
      state := 'IDLE' \+
      serviceRequest reset \+
      currentHandlerInformed := false
            if currentHandlerInformed
end-parallel
```

The precondition for this method is that the instance variable `currentHandlerInformed` should be true. This variable is an instance variable of the `Connection` class that cannot be modified by any client. The only statement that sets it to true is in the `p_informCommsProvider:` parallel method of the `Connection` class and one of the preconditions for that method is that the state should be equal to 'TERMINATING', as can be seen below:

[13] The SLOOP specification of the methods of the Connection class can be found in Appendix B, Section B.7.

```
"p_informCommsProvider method of the Connection class"
parallel
      commsAgent terminate: self cause: terminatingReason \+
      currentHandlerInformed := true
            if state = 'TERMINATING' and:
            [(terminatingReason = 'completed')
            and: [currentHandlerInformed not]]
end-parallel
```

Thus, the state is only set to 'IDLE' if `currentHandlerInformed` is true. In turn, `currentHandlerInformed` is only set to true if the state is 'TERMINATING'. This implies that the state is only set to 'IDLE' if it is currently in the 'TERMINATING' state. This concludes the informal correctness arguments for B2.1.

Thus, due to the **data encapsulation** provided by the SLOOP method, the instance variables of an object can only be modified by that object itself or by methods provided by that object to its clients. If a method is provided that allows clients to set the value of the instance variable under consideration (`state` in the above example) to the value of interest ('IDLE' in the above example), then all the statements of the program need to be checked to determine whether this method is being invoked by any of them. However, if no such method exists (as in the above example), then the statements that need to be examined are reduced to those appearing in the class definition itself. The procedures used during correctness reasoning in the SLOOP method therefore take advantage of the **structuring** and **data encapsulation** capabilities of object-orientation.

Part B2.2:

In this part it needs to be shown that the `ServiceProviderSimulator` instances are the only objects that invoke the `terminate:` method of the `Connection` class. Note that up until now the correctness arguments have all been presented from first principles. However, one of the advantages of an object-oriented method such as SLOOP is the fact that correctness arguments can also be **reused**. This is first illustrated in this part and then applied in all the remaining correctness arguments.

The only method that causes the connection state to change to 'TERMINATING' is the `terminate:` method of the `Connection` class. The only class that invokes the `terminate:` method of the `Connection` class at this level of refinement is the `ServiceProviderSimulator` class[14]. This is determined by inspecting the correctness properties of all the classes and their methods. It is found that the following property of the `p_generateEvent` method of the `ServiceProviderSimulator` class refers to the `terminate:` method:

```
"precedence property of the p_generateEvent method"
generatingEvent ∧ serviceRequest notNil ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧
      serviceRequest isNil ∧ ¬generatingEvent
                        "DP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the connection currently associated with the service request is terminated and that the service provider simulator becomes available to service a new service request."

[14] The ServiceProviderSimulator class is specified in Appendix B, Section B.13.

The `postconditions:withArguments:` construct was first discussed in Chapter 4, Section 4.3.4.2. To recapitulate: it is used when a method sends a message to another object and the postconditions of the method being executed by the other object have significance in the correctness properties of the sending method. Thus, in the case of the *DP1-01* property of the `ServiceProviderSimulator`, the postconditions of the `terminate:` method of the `Connection` class will hold in addition to the postconditions that will hold as a result of assignments to attributes of the `ServiceProviderSimulator` instance. These postconditions will hold if `'completed'` is used as the argument of the message.

The introduction of such a construct has several advantages. It **prevents** the problem of **inconsistency** which could arise if the same postconditions were specified in multiple places and it also **highlights the use of another method** in the property specification. This is particularly useful in correctness arguments such as the current one, where one is trying to identify all the methods that are invoking the `terminate:` method.

Note that it is not necessary to verify at this point that the `p_generateEvent` method indeed invokes the `terminate:` method. The properties of the `ServiceProviderSimulator` class are reused without proving them from first principles. The correctness of the methods of the `ServiceProviderSimulator` class are shown at the time when the correctness arguments for the `ServiceProviderSimulator` class itself are presented. Thereafter it can be assumed that the behaviour of this class is as specified by its correctness properties.

Part B2.3:

It now needs to be shown that the `ServiceProviderSimulator` instance only invokes the `terminate:` method if it has a service request assigned to it. This follows directly from the *DP1-01* property of the `p_generateEvent` method of the `ServiceProviderSimulator` class:

```
"precedence property of the p_generateEvent method"
generatingEvent ∧ serviceRequest notNil ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil ∧
      ¬generatingEvent  "DP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the connection currently associated with the service request is terminated and that the service provider simulator becomes available to service a new service request."

Part B2.4:

It has now been argued that a connection is only terminated by the service provider simulator and then only if it has a service request associated with it. It must be shown next that after the service request has been assigned to a service provider simulator, it is no longer present in the `inputQ`. The following precedence properties of the `CC_SimulationActivation` and `CC_Activation` classes respectively are relevant.

```
<∀ aServiceRequest where
      < ∃ aServiceProviderSimulator where
      spAgentContainer includes: aServiceProviderSimulator::
            aServiceProviderSimulator serviceRequest = aServiceRequest
      ::
         <∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
         > precedes
            aServiceProviderSimulator serviceRequest = aServiceRequest
      >
>                                    "AP2-01 (CC_SimulationActivation)"
```

"A service request is assigned to an element of the service provider container only if the former has been enqueued in a service queue and has remained in the queue until it was assigned to the service provider container element."

```
<∀ aServiceRequest where
      < ∃ aServiceQueue ::
            aServiceQueue includes: aServiceRequest
      ::
         inputQ includes: aServiceRequest
         precedes
            aServiceQueue includes: aServiceRequest
      >
>                                                    "AP2-02 (CC_Activation)"
```

"A service request is allocated to a service queue only if the service request has been enqueued in the inputQ and has remained in the latter until it was allocated to the service queue."

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest ensures
      ¬(inputQ includes: aServiceRequest) ∧
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest)
      >
>                                                    "AP1-05 (CC_Activation)"
```

"A service request remains in the inputQ until it is assigned to a service queue."

In the above properties `aServiceQueue` is quantified over all service queues associated with `ServiceCategory` instances in the `scContainer`. This quantification is not shown in order to make the property specifications less cluttered.

The argument is as follows: Property *AP2-01* specifies that a service request can only be assigned to a service provider simulator if it comes from a service queue. In turn, a service request can only be allocated to a service queue if it comes from the `inputQ` (property *AP2-02*). Furthermore, when a service request is allocated to a service queue, it is removed from the `inputQ` (property *AP1-05*). Since there is no statement which adds the service request to the `inputQ` while it is in the service queue or while it is assigned to a service provider simulator, it follows that the service request is not present in the `inputQ` while it is assigned to a service provider simulator.

There is no statement which adds a service request to the `inputQ` once it has been removed, except if the connection has been terminated and has reached the 'IDLE' state (from Part B1). Thus, a service request that is an element of the `inputQ` is no longer present in the `inputQ` by the time the state of the associated connection changes to 'IDLE'. This concludes Part B2 of the correctness arguments.

Thus, the above example has illustrated the following:

- **Location counters are not considered** during correctness reasoning.
- It is also **not necessary** to be concerned about the **allocation of statements to processors**. At the design level, correctness reasoning is in terms of the **parallel statements**, each of which executes atomically.
- **Data encapsulation** provides a mechanism to reduce the effort required during correctness reasoning.
- The **reuse** of correctness properties significantly reduces the correctness reasoning effort.

7.3.1.3 Using a global invariant property to discuss the responsibility of the client object regarding preconditions in correctness properties

This section focuses on the **responsibilities of the client object** in ensuring that **preconditions are satisfied** when a method is invoked. In Chapter 5, section 5.4.1.3, the following property was specified in order to ensure that a service request does not get overwritten by another one before its processing has been completed:

AS3-09. A service provider / service provider simulator services a single service request at a time.

Specifying the correctness property formally

In the `CC_SimulationActivation` class, this property is specified as follows:

```
<∀ aServiceProviderSimulator where
      spContainer includes: aServiceProviderSimulator ::
      <∀ aServiceCategory where
      scContainer includes: aServiceCategory ::
      invariant aServiceProviderSimulator serviceRequest notNil ⇒
            ¬aServiceProviderSimulator canAcceptNextSR:
            (aServiceCategory serviceQCategory)
      >
>                                          "AS3-09 (CC_SimulationActivation)"
      "A service provider simulator services a single service request at a time."
```

Thus, once a service request is allocated to a service provider simulator, no other service request can be allocated to that service provider simulator until the latter has completed servicing the first service request. It is therefore not possible to overwrite the first service request.

The strategy to be followed for the correctness arguments

Since this is an invariant, it has to be shown that the property holds initially (part A), as well as after the execution of each parallel statement (part B). There are two aspects to the arguments in part B. Firstly, it needs to be shown that the `ServiceProviderSimulator` instance does not accept a new service request if another service request is already assigned to it (Part B1). If it is found that the `ServiceProviderSimulator` class achieves this via the preconditions of the method which accepts a new service request for processing, then it also has to be shown that those preconditions are indeed met whenever that method is invoked by a client (Part B2).

This is because the above correctness property is not only required to hold for the `ServiceProviderSimulator` class, but also for the system as a whole. (Property *AS3-09* is a correctness property of the composite `CC_SimulationActivation` class.)

Since the behaviour of the target object is undefined if the preconditions are not met, it is imperative to check that the client only invokes the method under the correct circumstances. This is called the "demanding" design approach in [Meye97], where the target object expects its methods to be invoked only if their respective preconditions are met and where the methods of the target object do not contain code to take some action if the preconditions are not met.

The motivation for a "demanding" design approach is discussed at length in [Meye97]. It is argued that the client object is best equipped to deal with the cases where the preconditions cannot be met (often the target object cannot do anything more constructively than print an error message). The example in this section demonstrates the impact on correctness reasoning when this approach is adopted.

The correctness arguments

Part A:
Immediately after a `ServiceProviderSimulator` instance has been created and initialized, it does not have any service request assigned to it, as can been seen in the SLOOP specification of the `ServiceProviderSimulator` class in Appendix B, Section B.13. Property *AS3-09 (CC_SimulationActivation)* is therefore satisfied initially.

Part B:
It now has to be shown that the property holds after the execution of any parallel statement of the program. As discussed in the previous section, the magnitude of the task is reduced by the fact that only those statements that are relevant to this property need to be considered.

Part B1:
It is evident from the total correctness property of the `processServiceRequest:` method of the `ServiceProviderSimulator` class that the `ServiceProviderSimulator` class ensures that property *AS3-09* is not violated. (The parameter that is passed in the `processServiceRequest:` message is called `aServiceRequest` and it refers to the new service request.)

```
"Total correctness property of the processServiceRequest:
method"
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Thus, the precondition for accepting a new service request is that the `canAcceptNextSR:` method should return true. From the total correctness property of the latter it is clear that the `canAcceptNextSR:` method only returns true if no service request is assigned to that `ServiceProviderSimulator` instance (and the next service category to be serviced matches the one passed as parameter), as can be seen below:

```
"total correctness property of the canAcceptNextSR: method"
true results-in
      methodReturnValue = ((requestingServiceCategory =
      categoriesServed at: (categoryIndex + 1)) ∧
      (serviceRequest isNil))
                          "DL1-04 (ServiceProviderSimulator)"
```

A `ServiceProviderSimulator` instance therefore only accepts a new service request if it is not currently processing another one.

Part B2:
It is the responsibility of the client to ensure that `canAcceptNextSR:` returns true prior to invoking the `processServiceRequest:` method. If the client does not ensure that the precondition of the total correctness property (*DL1-06*) of `processServiceRequest:` holds, the `ServiceProviderSimulator` instance does not have any obligations regarding the correctness properties of the class and its methods. In the call centre example, the `ServiceProviderSimulator` class overwrites the existing service request in the

processServiceRequest: method if the preconditions are not met, as is evident from the statements of this method:

```
"The processServiceRequest: method:
sequential
newEventRequired := true \+
serviceRequest := aServiceRequest \+
categoryIndex := (categoryIndex + 1) \\ nrOfCategoriesServed
end-sequential
```

In the correctness arguments of property *AS3-09 (CC_SimulationActivation)* it is therefore necessary to show that the precondition will always hold when the processServiceRequest: method is invoked. The only statement which invokes the latter is the assignToSP: method of the ServiceCategory[15] class.

The following total correctness property applies to the assignToSP: method:

```
sr notNil ∧ availableServiceProvider notNil results-in
      methodReturnValue = self ∧
      availableServiceProvider
            postconditions:(#processServiceRequest:)
            withArguments: #(sr)  ∧
      sr postconditions: (#serviceProvider:)
            withArguments: #(availableServiceProvider)
                                          "DL1-10(ServiceCategory)"
```

where availableServiceProvider is defined in a macro as:

```
availableServiceProvider ≡
      spSubset detect: [:each | each canAcceptNextSR:
      serviceQCategory]
```

Thus, the ServiceCategory instance assigns the service request denoted by sr to the first service provider simulator in spSubset which can accept another service request. (The spSubset instance variable of the ServiceCategory class represents the set of service provider simulators that have the capability to process service requests that are enqueued in the serviceQ of this ServiceCategory instance.)

The processServiceRequest: message is therefore only sent to the ServiceProviderSimulator instance if the canAcceptNextSR: message to that instance has returned true. This concludes the informal correctness arguments regarding property *AS3-09 (CC_SimulationActivation)*.

This section has shown how a class can restrict its responsibilities regarding the preservation of a property by specifying the appropriate preconditions for its methods. It is then the **responsibility** of the **client** of these methods to ensure that the relevant preconditions are met when these methods are invoked.

7.3.1.4 Using an unless property to demonstrate how the correctness properties specified for the class itself as well as those inherited from its parent class are applied in correctness arguments

It is important to understand the effect of **inheritance** on correctness properties. There are three distinct cases:

- A correctness property may be **reused as is** in the descendant.
- A correctness property may be **added** in the descendant.
- A correctness property of the descendant may **override** one in its ancestor.

[15] The ServiceCategory class is specified in Appendix B, Section B.10.

The first two cases are covered in the first example described in this section. Subsequently, another example is given which demonstrates how a correctness property can be overridden by a descendant.

The following is one of the **unless** properties of the `ServiceProviderSimulator` class :

```
serviceRequest isNil ∧ ¬newEventRequired unless
      serviceRequest notNil ∧ newEventRequired
                              "AS4-01 (ServiceProviderSimulator)"
```

"When a new service request is assigned to the service provider simulator then a new service provider simulator event is required."

Property *AS4-01* specifies that when the value of the `serviceRequest` instance variable changes from nil to not nil, then the value of the `newEventRequired` instance variable changes from false to true. This is an example of a correctness property that is **added** in a descendant. The `EventSimulator` class, which is the parent of the `ServiceProviderSimulator` class[16], does not contain this property at all. However, it will now be shown how some of the properties defined for the parent class are **reused as is** in the correctness arguments for property *AS4-01(ServiceProviderSimulator)*.

The strategy to be followed for the correctness arguments

Since the `ServiceProviderSimulator` instance does not provide any methods to clients to modify the `serviceRequest` and `newEventRequired` instance variables, it is merely necessary to check the statements of the `ServiceProviderSimulator` class and its ancestors in order to verify that this property is not violated.

First of all it is shown in **part A** that when the value of `serviceRequest` changes from nil to not nil, then `newEventRequired` is set to true. Thereafter it is shown in **part B** that `serviceRequest` is only set to a non-nil value when the value of `newEventRequired` changes from **false** to true, i.e. `newEventRequired` is not already true when `serviceRequest` is set to a non-nil value.

Part B has two subsections:
B1) Firstly it is shown that `newEventRequired` is set to false and `serviceRequest` is set to nil upon initialization. Thus, when `processServiceRequest:` is executed the first time, `newEventRequired` is false, since no other method sets this variable to true. (It is in the `processServiceRequest:` method where `serviceRequest` is set to a non-nil value and where `newEventRequired` is set to false.)

B2) It must then be shown that whenever `processServiceRequest:` is invoked after that, `newEventRequired` has the value false. When `processServiceRequest:` is executed, `newEventRequired` is set to true and `serviceRequest` is set to a non-nil value. Since one of the preconditions of the `processServiceRequest:` method is that `serviceRequest` should be nil, it means that the `processServiceRequest:` method can only be executed again once `serviceRequest` has been set to nil. Note that it is not a precondition of this method that `newEventRequired` should be false. In order to be sure that `processServiceRequest:` is not invoked while `newEventRequired` is still true, it therefore has to be shown that `serviceRequest` will remain not nil for at least as long as `newEventRequired` is true.

[16] The ServiceProviderSimulator class is specified in Appendix B, Section B.13 and its parent class, the EventSimulator class, is specified in Appendix B, Section B.5.

Once `newEventRequired` is true, a timer will eventually be started, at which point `newEventRequired` is set to false. The `serviceRequest` variable is only set to nil once this timer has expired. This means that `newEventRequired` is set to false before `serviceRequest` is set to nil. Since `processServiceRequest:` is the only method which sets `newEventRequired` to true, it follows that `newEventRequired` will always be false when `serviceRequest` is nil, i.e. when `processServiceRequest:` is invoked. The sequence of events as dictated by the preconditions of the above methods is shown below in tabular format. (The way in which these preconditions determine the sequence of events is referred to as synchronisation constraints, a topic which was discussed in Chapter 3, Section 3.2.2.3.)

Step	Method executed	Variables modified, methods invoked
1	`processServiceRequest:`	`newEventRequired := true` `serviceRequest := aServiceRequest`
2	`p_simulate: timeoutEventsIn:`	`newEventRequired := false` `startRandomTimer:withMaximum`
3	`p_simulate:timeoutEventsIn:`	`generatingEvent := true` `resetTimerExpired:`
4	`p_generateEvent`	`serviceRequest := nil` `generatingEvent := false`

Table 7-1. Sequence of events involving the `newEventRequired` and `serviceRequest` instance variables.

The correctness arguments

PartA:
It needs to be shown that when the value of `serviceRequest` is set to not nil, then `newEventRequired` is set to true. There is only one method which sets the `serviceRequest` instance variable to a non-nil value, viz. `processServiceRequest:`. The total correctness property of this method specifies that when the value of `serviceRequest` changes from nil to not nil, then `newEventRequired` is set to true:

```
"Total correctness property of the processServiceRequest:
method"
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Property *DL1-06* therefore guarantees that when the value of `serviceRequest` changes from nil to not nil, then `newEventRequired` is set to true.

Part B:
It must now be shown that `newEventRequired` will change from **false** to true when the value of `serviceRequest` is set to a non-nil value.

The total correctness property of the `processServiceRequest:` method guarantees that `newEventRequired` is set to **true** when the method is **executed**, as was shown above. It therefore remains to be shown that `newEventRequired` will always be **false** when the `processServiceRequest:` method is **invoked**.

Part B1:
From the total correctness property of the `initialize` method of `EventSimulator`, the parent class, it is clear that `newEventRequired` is false initially.

```
"Total correctness property of the initialize method"
true results-in methodReturnValue = self ∧
rand notNil ∧ newEventRequired = false ∧
currentRandomTimeoutValue = 1 ∧
generatingEvent = false ∧
timerOutstanding = false                    "DL1-01 (EventSimulator)"
```

The `moreInit:using:` method of the `ServiceProviderSimulator` class sets `serviceRequest` to nil initially as can be seen from the total correctness property of that method.

```
"Total correctness property of the moreInit:using: method"
true results-in methodReturnValue = self ∧
      serviceRequest isNil ∧
      aConfiguration postconditions: (#assignSPCategory) ∧
      serviceProviderCategory notNil ∧
      categoriesServed notNil ∧
      self postconditions: (#registerServiceProvider: using:)
            withArguments: #(scContainer aConfiguration)
                        "DL1-01 (ServiceProviderSimulator)"
```

Since both these methods are executed when the `ServiceProviderSimulator` instance is created, `newEventRequired` is false and the value of `serviceRequest` is nil when the `processServiceRequest:` method is invoked the first time. This is because `processServiceRequest:` is the only method which sets `newEventRequired` to true and `serviceRequest` to not nil. It now remains to be shown that `newEventRequired` will be false whenever `processServiceRequest:` is executed thereafter.

Part B2:
The safe liveness property of the `p_simulate:timeoutEventsIn:` method of the `EventSimulator` class (i.e. the **parent** class) ensures that `newEventRequired` will eventually become false, as can be seen below:

```
newEventRequired ensures
      self postconditions: (#startRandomTimer:withMaximum:)
      withArguments:
      #(aTimerServices (aTimerServices maximumTimeout))
      ∧ ¬newEventRequired              "DP1-01(EventSimulator)"
```

"When newEventRequired is true, it ensures that a simulation timer is started and newEventRequired becomes false."

Since the `processServiceRequest:` method is the only one that sets `newEventRequired` to true, `newEventRequired` will remain false until `processServiceRequest:` is executed again. The latter is not executed while `serviceRequest` has a non-nil value. One therefore has to show next that `serviceRequest` will only be set to nil once `newEventRequired` has already been set to false.

Property *DP1-01(ServiceProviderSimulator)* describes the conditions under which `serviceRequest` is set to nil. It specifies that it will only happen if `generatingEvent` is true, as can be seen below:

```
generatingEvent ∧ serviceRequest notNil ensures
        (serviceRequest connection) postconditions: (#terminate:)
        withArguments: #('completed') ∧
        serviceRequest isNil ∧ ¬generatingEvent
                                "DP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the connection currently associated with the service request is terminated and that the service provider simulator becomes available to service a new service request."

As is evident from property *DP1-02* of the `EventSimulator` **parent** class, `generatingEvent` is only set to true if a timer started by `self` expires. In this case `self` is the current `ServiceProviderSimulator` instance.

```
self timerExpired: timerEventQ ensures
        generatingEvent ∧
        self postconditions: (#resetTimerExpired:)
        withArguments: #(timerEventQ) "DP1-02(EventSimulator)"
```

"When a simulation timer expires, it ensures that generatingEvent becomes true."

The total correctness property of the `timerExpired:` method of the `EventSimulator` **parent** class specifies that it will only return true if the `timerEventQ` contains an element which has `self` as the `timeoutRequestor`, i.e. the current `ServiceProviderSimulator` instance requested the timer earlier on.

```
"Total correctness property of the timerExpired: method"
true results-in methodReturnValue =
        (timerEventQ detect: [:each |
        each timeoutRequestor == self ]
        ifNone: [nil]) notNil          "DL1-03(EventSimulator)"
```

Thus, `generatingEvent` is **only set to true upon the expiry of a timer** started by the current `ServiceProviderSimulator` instance. Property *DP1-01* of the `EventSimulator` **parent** class guarantees that `newEventRequired` is set to false when an `EventSimulator` subclass requests a timer to be started, as can be seen below.

```
newEventRequired ensures
        self postconditions: (#startRandomTimer:withMaximum:)
        withArguments:
        #(aTimerServices (aTimerServices maximumTimeout))
        ∧ ¬newEventRequired            "DP1-01(EventSimulator)"
```

"When newEventRequired is true, it ensures that a simulation timer is started and newEventRequired becomes false."

Thus, the expiry of a timer has to be preceded by the setting of that timer, at which point `newEventRequired` is set to false. The value of `serviceRequest` can therefore only be set to not nil once `newEventRequired` has been set to false.

The correctness arguments for part B are summarised as follows:
In order to show that `newEventRequired` is always false when `processServiceRequest:` is executed, it was first shown that `newEventRequired` is set to false and `serviceRequest` is set to nil upon initialization. Thus, when `processServiceRequest:` is executed the first time, `newEventRequired` is false, since no other method sets this variable to true.

Thereafter, the following occurs: When `processServiceRequest:` is executed, `newEventRequired` is set to true and `serviceRequest` is set to a non-nil value. Eventually, a timer is started, at which point `newEventRequired` is set to false. Once the timer expires, `generatingEvent` is set to true. Only once `generatingEvent` is true, can

`serviceRequest` be set to nil. Since `generatingEvent` is only set to true upon expiry of a timer, it follows that once `processServiceRequest:` has been executed, `newEventRequired` is always set to false before `serviceRequest` is set to nil. This concludes the correctness arguments for property *AS4-01*.

In order to reason about the correctness of the behaviour of the `ServiceProviderSimulator` class, it is necessary to take the correctness properties of its parent class, `EventSimulator`, into account. The above example has therefore demonstrated how correctness properties that are **inherited** from a **parent** class are **reused** in the correctness arguments of properties of a **descendant** class.

Overriding a property of an ancestor

The `CC_SimulationActivation` class provides an example of where a correctness property **overrides** a property of an ancestor. The `initCommsAgent` method of the `CC_Activation` class has the following total correctness property:

```
"Total correctness property of the initCommsAgent method"
true results-in methodReturnValue notNil
                                        "DL1-05 (CC_Activation)"
```

The `CC_SimulationActivation` subclass **overrides** the `initCommsAgent` method and the above property is **specialized** in the subclass in the following way:

```
"Total correctness property of the initCommsAgent method"
true results-in methodReturnValue notNil ∧
      CC_SimulationInterfacesPkg:::CommsProviderSimulator
      postconditions: (#startSimulation)
                                        "DL1-05 (CC_Activation)"
```

Thus, the `CC_Activation` class merely specifies that the method will return a non-nil value, but it does not specify which class should be instantiated. That is left up to the subclass.

The overriding of a property is indicated syntactically by repeating the number of the property from the ancestor and including the name of the ancestor in brackets. This is similar to the way in which a method in a descendant overrides a method with the same name in the ancestor. (All properties that are not inherited are assigned identifiers that are unique within that class.)

In the above example the preconditions are left unchanged, while the **postconditions** are **strengthened** in the **subclass**. It is important that the designer should take into account that **preconditions** may **not** be **strengthened** and **postconditions** may **not** be **weakened** when correctness properties are overridden in a subclass. This is to ensure that a class could be replaced with its subclass while guaranteeing that all the properties that used to hold for the parent class will still hold when the subclass is used instead [Meye97].

In order to ensure that preconditions can be weakened and postconditions can be strengthened, care should be taken during the specification of properties to avoid overspecification. For example, in the `TimerServices`[17] class, the size of the `timeoutCollection` array is dependent on the maximum timeout value (it is equal to `maximumTimeout + 2`). It is important that all other properties that depend on the size of the `timeoutCollection` array should refer to '`timeoutCollection size`', rather than '`maximumTimeout + 2`', since the size of the `timeoutCollection` array might be calculated differently in subclasses of

[17] The TimerServices class is described in Appendix B, Section B.11. Details about its design and its properties are also given in Section 7.3.2.2.

`TimerServices`. For such subclasses it should not be necessary to modify the properties of the methods that use the size of the `timeoutCollection` array.

For example, the clean behaviour property of the `start:id:for:` method should not have to change if the size of the `timeoutCollection` array is calculated differently.

```
invariant    1 ≤ writeIndex ∧ writeIndex ≤ timeoutCollection size
                                        "DS3-02(TimerServices)"
```

The above examples have demonstrated how a correctness property first defined in a parent class can be **reused as is** in a descendant class. It has also been shown how correctness properties can be **added** in descendant class. The third type of reuse allowed for the **specialization** of a correctness property in a descendant class, provided the modifications adhere to the rule given earlier in this section. This concludes the discussion about the impact of **inheritance** on correctness arguments.

7.3.2 Liveness properties

This section deals with three different types of liveness properties, viz. total correctness, intermittent assertions and responsiveness properties. The correctness arguments for examples of these properties are used as a vehicle to discuss various aspects of the correctness reasoning in the SLOOP method. The first example emphasizes the atomicity of sequential methods and how this characteristic can be used in correctness properties. The intermittent assertion example shows how the computational model is used to reason about progress and the responsiveness example demonstrates the importance of showing that preconditions will eventually hold when correctness properties are being reused in correctness arguments of liveness properties.

7.3.2.1 Showing why the postconditions of a total correctness property can be used in an ensures relation

The sequential methods in a SLOOP program are either executed as part of the sequential statements in the *activation-section* or they are invoked from within parallel statements. Since each parallel statement is executed **atomically** (i.e. **statement interleaving** takes place at the level of **parallel** statements), concurrency does not impact on sequential statements embedded in parallel statements.

The safe liveness property of the `p_doWrapUp` method of the `Connection` class demonstrates how the postconditions of a sequential method can be used in an **ensures** relation. Recall that the **ensures** relation only applies if the postcondition of the relation is reached via the execution of a **single** SLOOP statement. Although the `reset` method of the `ServiceRequest` class that is invoked by the `p_doWrapUp` method of the `Connection` class contains multiple sequential statements, they are embedded in the single parallel statement of the `p_doWrapUp` method, as shown below:

```
"p_doWrapUp method of the Connection class"
message pattern p_doWrapUp
method properties
"Safe liveness"
currentHandlerInformed ensures
state = 'IDLE' ∧ serviceRequest postconditions: (#reset) ∧
¬currentHandlerInformed                    "DP1-02(Connection)"
parallel
state := 'IDLE' \+
serviceRequest reset \+
currentHandlerInformed := false
      if currentHandlerInformed
end-parallel
```

```
"reset method of the ServiceRequest class"
message pattern reset
method properties
"Total correctness"
true results-in methodReturnValue = self ∧ serviceQ isNil ∧
      serviceRequestCategory isNil ∧ serviceProvider isNil ∧
      categorisationData isNil       "DL1-12(ServiceRequest)"
sequential
serviceQ := nil
[] serviceRequestCategory := nil
[] serviceProvider := nil
[] categorisationData := nil
end-sequential
```

Thus, despite the fact that the `reset` method contains four separate SLOOP statements, each one achieving part of the postcondition, the client of a sequential method views the execution of such a method as an atomic event. The postconditions of a sequential method may therefore be used in the postconditions of an **ensures** relation. In this example it is therefore guaranteed that no other parallel statements can be executed while the `Connection` instance and its associated `ServiceRequest` instance are being reset.

The **correctness arguments** to reason about a total correctness property of a sequential method proceed as for any **conventional sequential program, but taking the semantics of SLOOP sequential statements into account**. For example, if a statement contains a *conditional-component-part-list*[18] as in the `registerServiceProvider:using:` method of the `ServiceProviderSimulator` class below, then all the *component-parts* of the list are subject to the associated condition.

The order of the statements is significant. For example, in the code fragment below it is clear that the statement setting `nrOfCategoriesServed` to zero has to be executed before the loop that follows. This is because this variable is incremented within the loop whenever the simulator is registered with a service category. Once the loop has been completed, this variable contains the number of service categories serviced by this simulator.

If another method is invoked by any of the statements, then the semantics for the invocation are as for a function call in a conventional programming language. The total correctness property of the `registerServiceProvider:using:` method below specifies the conditions that should hold when the method is entered and it also specifies the conditions that should hold when control exits from the method.

```
message pattern registerServiceProvider: scContainer
            using: aConfiguration
"Registers the ServiceProviderSimulator with the relevant
service categories"
method macros
maxCategories ≡ aConfiguration maximumServiceCategories
method properties
"Total correctness"
true results-in methodReturnValue = self ∧
      <∀aServiceCategory where
      scContainer includes: aServiceCategory ∧
      aServiceCategory servicedBy: serviceProviderCategory ::
```

[18] The *conditional-component-part-list* was defined in Chapter 4, Section 4.3.6.2.

```
        aServiceCategory postconditions: (#addSP:)
        withArguments: #(self) ∧
        categoriesServed includes:
              (aServiceCategory serviceCategory)
        > ∧
        nrOfCategoriesServed = categoriesServed size ∧
        categoryIndex ≥ 0  "DL1-05 (ServiceProviderSimulator)"
sequential
  nrOfCategoriesServed := 0
[] < [] j where 1≤j≤maxCategories ::
(scContainer at: j) addSP: self \+
nrOfCategoriesServed := nrOfCategoriesServed + 1 \+
categoriesServed addLast:((scContainer at: j) serviceCategory)
      if (scContainer at: j) servicedBy: serviceProviderCategory
>
[] categoryIndex := 0
end-sequential
```

Since there can be **no interference** while the statements of a **sequential method** are executing, it is guaranteed that once the condition of a conditional statement within the sequential method has been evaluated, its value cannot change before the rest of the statement starts executing.

The examples in this section have highlighted the way in which the execution of sequential methods should be interpreted, i.e. the method executes as an **atomic unit**. No concurrency needs to be taken into account when reasoning about the behaviour of the statements within a sequential method. As far as the computational model is concerned, a sequential method is executed as part of a **single parallel** statement. The total correctness properties of a sequential method can therefore be used in an **ensures** relation (and in the other SLOOP relations as well).

7.3.2.2 Using an intermittent assertion property to demonstrate how the repeated execution of parallel statements guarantees progress, provided the preconditions hold at some point

The aim in this section is to demonstrate how the **repeated execution of the parallel statements** selected for the program **eventually results in the postconditions** specified by the **liveness** properties, provided that their preconditions hold at some point. The example is taken from the list of intermittent assertion properties specified in Chapter 5:
AL2-01.Once a timer has been started, it will eventually expire or it will be stopped.

Specifying the correctness property formally

In order to write this property more formally, it is necessary to consider how timers are represented in the system. A brief description of the design of the `TimerServices` class was presented in Chapter 6, Sections 6.2.4 and 6.3.1, and full details are given in Appendix B, Section B.11. However, for convenience, a short summary is presented here.

The `TimerServices` instance, which handles all timer requests, creates a `TimeoutElement` instance whenever a timer is started. In order to be able to inform the requestor of the expiry of the timer, the `TimerServices` instance determines when the timer would expire and then stores the `TimeoutElement` instance in the list of timers that will expire at the calculated time. These **lists** are stored in an instance of the `Array` class, called `timeoutCollection`.

The `TimerServices` class implements this array as a **circular** array. Each position in the array represents one second. The `TimerServices` instance maintains an index into the array. This

index is called `currentTick` and is advanced every second (it is incremented modulo the size of the array). Thus, the entry in the array which will be reached x seconds from the current moment can be calculated using the value of `currentTick`, the size of the array and the value of x. Each entry in the array is an ordered collection of `TimeoutElement` instances.

When a timer expires, its associated `TimeoutElement` instance is removed from the list in `timeoutCollection` and entered into `timerEventQ`, which is checked by all timer requestors on a regular basis. Appendix B, Section B.11, contains two diagrams (Figures B-3(a) and B-3(b)) illustrating the above concepts.

The responsibility for ensuring that each timer will expire unless it is stopped, lies with the `TimerServices` class. When property *AL2-01* is specified more formally, it can therefore be written in terms of the instance variables of the `TimerServices` class:

```
<∀ aTimeoutElement where
      <∃ i where 1 ≤ i ≤ (timeoutCollection size) ::
      (timeoutCollection at: i) includes: aTimeoutElement
      > ::
      ¬aTimeoutElement timerServicesCompleted leads-to
            aTimeoutElement timerServicesCompleted
>                                          "DL2-01(TimerServices)"
```

"Once a timer has been started, i.e. it is present in one of the lists associated with timeoutCollection, the TimerServices instance will eventually complete its responsibilities regarding the timer (i.e. the timer will either be stopped or the TimerServices instance will indicate its expiry to the requestor of the timer)."

The strategy to be followed for the correctness arguments

It needs to be shown that once a `TimeoutElement` instance has been entered in one of the lists reached via the `timeoutCollection` array, the `TimeoutElement` instance will eventually be removed from the relevant list because the timer has been stopped by a client of the `TimerServices` instance or it will be removed because the timer has expired, in which case it is added to the `timerEventQ` list. Abnormal conditions are not considered at this level of abstraction, therefore the case where the timer could be stopped is ignored in this discussion. (At this level of abstraction the `TimerServices` class does not export a method to stop a timer.) The strategy for the correctness arguments is therefore as follows:

First of all it needs to be shown that the index which identifies the list of expired timers (the `readIndex`) will eventually reach the list containing the specified `TimeoutElement` instance (part A). It then needs to be shown that once that happens, the `TimeoutElement` instance will eventually be removed from the list and added to the `timerEventQ` (part B). Since progress is achieved via the infinitely often execution of parallel statements, the obvious place to start is at the parallel method defined for the `TimerServices` class, namely `p_runTimer:`. This method contains three parallel statements, as seen below:

```
parallel
   currentTime := SmalltalkLibPkg:::Time now asSeconds      "S1"
[] lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 and: [currentTimeoutElement isNil] "S2"
[] timerEventQ addLast: currentTimeoutElement \+
   currentTimeoutElement updateEndTime \+
   currentTimeoutElement timerServicesCompleted: true \+
   (timeoutCollection at: readIndex) removeFirst
      if currentTimeoutElement notNil                       "S3"
end-parallel
```

The purpose of each statement is summarised here, with more detail to follow. The first statement (*S1*) updates the `currentTime` instance variable with the current time (in seconds) whenever the statement is executed. The second statement (*S2*) updates two instance variables, viz. `lastTime` and `currentTick`, whenever at least one second has expired and all the `TimeoutElement` instances in the list identified by the current `readIndex` have been removed. The `lastTime` instance variable records the last time when `currentTick` was updated and is used to determine whether one second has already expired since the last update. Two *macro-variables*, viz. `difference` and `currentTimeoutElement` are used in this statement. They will described in more detail later. The last statement (*S3*) is used to remove the first `TimeoutElement` instance from the list identified by the `readIndex` and to add it to `timerEventQ`. The correctness arguments for property *DL2-01 (TimerServices)* will refer to the above statements repeatedly.

The correctness arguments

Part A:
The `readIndex` *macro-variable* is defined as follows in the `p_runTimer:` method of the `TimerServices` class (the one has to be added because SLOOP array indices start at one, not zero):

```
readIndex ≡ currentTick + 1
```

Since the `readIndex` is defined in terms of `currentTick`, it needs to be shown that `currentTick` will eventually be advanced from its current position, regardless of where that position is. The following statement (*S2*) in the `p_runTimer:` method advances the position of `currentTick`:

```
[] lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 and: [currentTimeoutElement isNil]   "S2"
```

The value of `currentTick` is incremented modulo the size of the `timeoutCollection` array, which implies the following:

```
invariant 0 ≤ currentTick ∧ currentTick ≤ (timeoutCollection size) - 1.
```

The advancement of `currentTick` depends on two conditions, involving the `difference` and `currentTimeoutElement` *macro-variables* respectively. These variables are defined as follows:

```
difference ≡ currentTime - lastTime
                  if (currentTime - lastTime) ≥ 0 ~
               currentTime + (86400 - lastTime)
                  if (currentTime - lastTime) < 0
[] currentTimeoutElement ≡
   (timeoutCollection at: readIndex) first
      if (timeoutCollection at: readIndex) isEmpty not ~
   nil
      if (timeoutCollection at: readIndex) isEmpty
```

First of all it needs to be shown that `difference` will eventually be greater than or equal to one. As can be seen from the above, `difference` is used to calculate the elapsed time since `currentTick` was last updated. It takes care of the rollover at midnight. The `currentTime` instance variable is updated via the following parallel statement of the `p_runTimer:` method.

```
currentTime := SmalltalkLibPkg:::Time now asSeconds      "S1"
```

Since statement *S1* must be executed **infinitely often**, it follows that `currentTime` will be updated infinitely often and it will therefore **eventually** result in `difference` having a value greater than or equal to one.

The second condition that has to be satisfied before `currentTick` and `lastTime` will be updated is that `currentTimeoutElement` should be nil. That happens only if the list of `TimeoutElement` instances is empty, as is evident from the *macro-definition* above. This will **eventually** become true as a result of the **infinitely often** execution of the third parallel statement of the `p_runTimer:` method. Each time that statement executes, it removes the first element of the list identified by `readIndex`, provided the list is not empty. Furthermore, the value of `readIndex` cannot change (being defined in terms of `currentTick`) until the condition for the execution of statement *S3* ceases to hold.

Thus, both conditions of statement *S2* of the `p_runTimer:` method will eventually become true, the first as a result of the infinitely often execution of statement *S1* and the second as a result of the infinitely often execution of statement *S3* of that method. Since statement *S2* is also executed infinitely often, it will eventually be executed when both conditions are true, in which case `lastTime` and `currentTick` will be updated. Thus, whatever the current value of `currentTick`, it will eventually be incremented modulo the size of the `timeoutCollection` array. Since `readIndex` is defined in terms of `currentTick`, `readIndex` will eventually identify the next list of `TimeoutElement` instances. Thus, regardless of the index of the list containing the `TimeoutElement` instance specified in property *DL2-01 (TimerServices)*, `readIndex` will eventually be equal to that index. This concludes the correctness arguments for Part A.

Part B:
It now remains to be shown that once the `readIndex` identifies a particular list of `TimeoutElement` instances, then all of those instances will eventually be removed and added to the `timerEventQ`. This follows vacuously from the infinitely often execution of statement *S3* of the `p_runTimer:` method. The presence of a `TimeoutElement` instance in the `timerEventQ` indicates to the corresponding timer requestor that the timer has expired. This concludes the correctness arguments of property *DL2-01 (TimerServices)*.

This section has focussed on the role of **parallel** statements in the correctness arguments for **progress** properties. Note that all parallel statements are executed repeatedly and in any order. If the effect of the statement is conditional, the execution of a statement may not have an effect each time it is executed.

The execution scenario as depicted in Table 7-2 shows one possible execution sequence (in this case statement *S1* is executed more often than the other two). The first statement of the `p_runTimer:` method always has an effect (it is not a conditional statement). The second statement in that method only has an effect if at least one second has expired since `currentTick` has been updated and if the list identified by `readIndex` is empty. This might not happen each time the statement is executed, as seen in the Table 7-2, where it is only updated when it is executed for the fourth time.

Statement executed	Effect
S2	`difference < 1` and there are still two elements in the list identified by `readIndex`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S3	One element is removed from the list identified by `readIndex` and added to `timerEventQ`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S2	`difference < 1` and there is still one element in the list identified by `readIndex`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S3	One element is removed from the list identified by `readIndex` and added to `timerEventQ`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S2	There are no elements in the list identified by `readIndex`, but `difference < 1`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S3	No action is taken.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S2	`difference is now ≥ 1` and the list identified by `readIndex` is empty, therefore `currentTick` is advanced.

Table 7-2. Parallel statement execution scenario.

In this section the role of the computational model in the correctness arguments for progress properties was described. The purpose was to show how the infinitely often execution of the parallel statements eventually results in the desired outcome. This is relatively simple if all the parallel statements are unconditional. If some are conditional, however, it is also necessary that the relevant conditions should eventually become true.

In the above example it had to be shown that the conditions of parallel statement (*S2*) would eventually become true. Since the first condition depended on the infinitely often execution of parallel statement *S1*, an **unconditional** parallel statement in the `p_runTimer:` method, it was trivial to show that it would eventually become true. The second condition depended on the infinitely often execution of statement *S3*. Although statement *S3* is a conditional statement, the condition which will prevent it from executing (i.e. when there are no more elements left in the list at the `readIndex` position) is exactly the condition that is required to facilitate the execution of statement *S2*. Thus, as long as there are elements in the list at that `readIndex` position, statement *S3* will execute and remove the first element whenever it is selected for execution (which is infinitely often). Eventually this list will become empty (since no elements can be added to the list at `readIndex`[19]), thereby satisfying the second condition of statement *S2*.

[19] This is guaranteed by invariant DS3-03 (TimerServices), which is as follows:
invariant writeIndex ~= readIndex

This section has described the correctness arguments for a liveness property from first principles. In the next section it is shown how the results of other correctness properties can be reused in the correctness arguments of a liveness property. The importance of showing that the preconditions will eventually hold is also described. This is analogous to the importance of showing that the conditions of conditional parallel statements will eventually hold in the above example.

7.3.2.3 Using a responsiveness property to demonstrate the importance of showing that the preconditions will eventually hold when reusing other correctness properties in the correctness arguments of a liveness property

In this section the focus is on the **reuse** of correctness properties and the **role of preconditions** in correctness arguments for **liveness** properties. It also shows how the concept of **eventuality chains** is applied in informal liveness property proofs.

In the call centre example, the responsiveness property is a very important one, since it is the property which ensures that the service user experiences the desired effect, i.e. it ensures that a service request is eventually serviced once the user has connected to the call centre. The SLOOP specification of the call centre in Appendix B is used as the basis for the discussions in this section. The level of abstraction of that specification assumes that service users will not abort service requests and the call centre will only receive service requests that belong to categories supported by the call centre. Service providers may be idle or busy, but not completely unavailable. The responsiveness property specified for the call centre system in Chapter 5, Section 5.4.2.3, is given below for the level of abstraction used in Appendix B.

AL3-01. Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of the service provider container.

Specifying the correctness property formally

In order to specify the above property more formally, one needs to determine what predicate holds for the service request associated with the connection immediately after a service user has connected to the call centre. In the `CallCentreSimulation` program, a service user trying to establish a connection is simulated by setting the `generatingEvent` instance variable of the `CommsProviderSimulator` class to true. If an idle `Connection` instance is available, it is assigned to the service user and the associated service request is entered into the `inputQ`. This is evident from the safe liveness property of the `p_generateEvent:target:` method of the `CommsProviderSimulator` class (the `newEventRequired` instance variable is set to true in order to trigger another simulated connection request from a service user after a random timeout):

```
"Safe liveness"
generatingEvent ∧ idleConnection notNil ∧
¬(inputQ includes:(idleConnection serviceRequest)) ensures
      ¬generatingEvent ∧ newEventRequired ∧
      inputQ last = (idleConnection serviceRequest) ∧
      idleConnection postconditions: (#assign)
                                    "AP1-01(CommsProviderSimulator)"
"If an event has to be generated and the maximum number of
connections have not yet been established, the communication
provider simulator ensures that a new connection is established,
the associated service request is appended to the input queue
and a new communication provider simulator event is again
required."
```

The *macro-variable* `idleConnection` used in the above property is defined as:

```
idleConnection ≡ self getIdleConnection: userConnections
```

The `getIdleConnection:` method returns the first idle connection that can be found in the `userConnections` array, or it returns nil if there are no idle connections. This behaviour can be deduced from the total correctness properties of the `getIdleConnection:` method of the `CommsProviderSimulator` class and the `isIdle` method of the `Connection` class listed below:

```
"Total correctness property of the getIdleConnection: method"
true results-in methodReturnValue =
userConnections detect: [:each | each isIdle] ifNone: [nil]
                                        "DL1-03(CommsProviderSimulator)"

"Total correctness property of the isIdle method"
true results-in methodReturnValue = (state = 'IDLE')
                                                "DL1-05(Connection)"
```

The `serviceRequest` method of the `Connection` class returns the service request associated with the connection. The `inputQ last = (idleConnection serviceRequest)` predicate of property *AP1-01(CommsProviderSimulator)* therefore specifies that the service request associated with the first idle connection is appended to the `inputQ`. The `idleConnection postconditions: (#assign)` predicate indicates that the `assign` method is invoked on the first idle connection. That method sets the `state` of the `Connection` instance to 'CONNECTED'. The interested reader is referred to Appendix B, Section B.7 for the specification of the total correctness properties of these methods.

The fact that property *AP1-01(CommsProviderSimulator)* contains an **ensures** relation, implies that at the time when the `generatingEvent` value is changed from true to false (simulating the successful connection of the service user to the call centre), a `Connection` instance is assigned to the service user and the associated service request is entered into the `inputQ` using a **single atomic** parallel statement. Thus, when a service user connects successfully to the call centre, the service request associated with the connection is added to the `inputQ`. This provides the necessary information to specify property *AL3-01 (CC_SimulationActivation)* more formally:

```
<∀ aConnection where userConnections includes: aConnection ::
inputQueue includes: aConnection serviceRequest leads-to
      <∃ aServiceProviderSimulator where
      spAgentContainer includes: aServiceProviderSimulator ::
      aServiceProviderSimulator serviceRequest =
            aConnection serviceRequest
      >
>                                   "AL3-01 (CC_SimulationActivation)"
```

"Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of the service provider container."

Note that the above property does not specify a unique association between the service requests in the `inputQ` and the service provider simulators. For example, if there are 5 service requests in the `inputQ`, then it is not necessary to have 5 service provider simulators. One simulator would suffice. The above property merely specifies that each service request in the `inputQ` will eventually be assigned to a service provider simulator.

The strategy to be followed for the correctness arguments

The above responsiveness property can be derived in two steps:

A) by applying the transitivity rule on the **leads-to** relation of the safe liveness properties that describe the sequence of events from the time that the connection is accepted by the call centre until the associated service request is assigned to a service provider, and
B) by showing that the preconditions of each of these properties will eventually become true.

The correctness arguments

Part A:
Properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)*, first presented in Chapter 5, Section 5.4.3.1, describe the path followed by a service request through the system. It has to be shown that a service request which is present in the `inputQ` is eventually processed by a service provider simulator. In this part, it is assumed that the preconditions of properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* will eventually hold.

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest ensures
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest)
      >
>                                          "AP1-05 (CC_Activation)"
```
"A service request remains in the inputQ until it is assigned to a service queue."

```
<∀ aServiceRequest ::
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
      >
      ensures
            < ∃ aServiceProviderSimulator where
            spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator.serviceRequest = aServiceRequest
            >
>                                          "AP1-06 (CC_Activation)"
```
"A service request remains in a service queue until it is allocated to an element of the service provider container."

In properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* `aServiceQueue` is quantified over all the service queues associated with `ServiceCategory` instances in the `scContainer`. This quantification is not shown in order to make the property specifications less cluttered.

In Chapter 4, Section 4.3.4.4, it was stated that the SLOOP **leads-to** relation can be derived by applying the same inference rules as specified for the UNITY **leads-to** relation. Those inference rules were presented in Chapter 2, Section 2.5.5. The correctness arguments for property *AL3-01(CC_SimulationActivation)* uses the first two inference rules, viz.

❑
$$\frac{p \text{ ensures } q}{p \rightarrow q}$$

❑
$$\frac{p \rightarrow q,\ q \rightarrow r}{p \rightarrow r} \quad \text{(transitivity)}$$

The **ensures** relation in the properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* can therefore be replaced with **leads-to** relations. Applying the transitivity rule on the **leads-to** relations in these properties allows one to deduce the following:

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest leads-to
      < ∃ aServiceProviderSimulator where
      spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator serviceRequest = aServiceRequest
      >
>
```

Thus, once a service request is present in the `inputQ`, it will eventually be serviced by a service provider simulator. The above example demonstrates how a **chain of eventualities** is used in the correctness arguments of property *AL3-01 (CC_SimulationActivation)*. The concept of proof by eventuality chains is described in [MaPn81b] as an approach "based on establishing a chain of eventualities that by transitivity leads to the ultimate establishing of the desired goal".

The results of properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* are reused here. Their correctness must be shown separately. As an example, the correctness arguments for property *AP1-06 (CC_Activation)* are presented in Section 7.3.3.1. This concludes part A of the correctness argument.

Part B:
It now remains to be shown that the preconditions of properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* will eventually hold.

The precondition of property *AP1-05 (CC_Activation)* specifies that the service request associated with a **new** connection should be present in the `inputQ`. It is evident from safe liveness property *AP1-01* of the `p_generateEvent:target:` method of the `CommsProviderSimulator` class, which was given at the beginning of this section, that the associated service request is entered into the `inputQ` when a new connection is established. The precondition of property *AP1-05(CC_Activation)* therefore holds once the user has connected successfully to the service centre, which means that its postcondition will eventually hold.

The precondition of property *AP1-06 (CC_Activation)* specifies that the service request has to be entered into a service queue. This follows directly from the postcondition of property *AP1-05 (CC_Activation)*. This concludes the second part of the correctness argument.

In this section it has been demonstrated how the properties of a **leads-to** relation can be utilised in the correctness arguments of a liveness property. It has illustrated the application of **eventuality chains** in informal liveness property proofs. It has also demonstrated that when reusing other correctness properties in the correctness arguments of a **liveness** property, it is important to show that the preconditions of the **properties** being reused will eventually become true. It is only if the preconditions do eventually become true that the postconditions can hold and that progress can take place. Preconditions play an equally significant role when the `postconditions:` construct is used in a correctness property. That is the topic of the Section 7.3.3.1, which covers the correctness arguments of one of the precedence properties. The next section discusses various precedence properties.

7.3.3 Precedence properties

In Chapter 5 three types of precedence properties were listed, viz. safe liveness, absence of unsolicited response and fair responsiveness. Informal correctness arguments are now given for examples of each of these correctness property types. The examples have been chosen to highlight various aspects of correctness reasoning. The safe liveness property example illustrates the usage of the `postconditions:` and `postconditions:withArguments:` constructs in correctness arguments. The absence of unsolicited response example is used to demonstrate how the distinctive characteristics of an **ensures** relation can be used in correctness arguments. Finally, the fair responsiveness example shows how correctness arguments are used to check that the classes selected for a system indeed satisfy the correctness properties as specified for the system under development.

7.3.3.1 Using a safe liveness property to highlight the impact of the postconditions: and postconditions:withArguments: constructs on correctness arguments

The purpose of this section is to describe the impact of **postconditions:** and **postconditions:withArguments:** constructs on correctness arguments. The discussion highlights the importance of showing that the **preconditions** of the methods referenced in the `postconditions:` and `postconditions:withArguments:` constructs are satisfied. Property *AP1-06 (CC_Activation)* is used in this example.

```
<∀ aServiceRequest ::
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
      >
      ensures
            < ∃ aServiceProviderSimulator where
            spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator serviceRequest = aServiceRequest
            >
>                                          "AP1-06 (CC_Activation)"
```

"A service request remains in a service queue until it is allocated to an element of the service provider container."

In property *AP1-06(CC_Activation)* `aServiceQueue` is quantified over all service queues associated with `ServiceCategory` instances in the `scContainer`. This quantification is not shown in order to make the property specifications less cluttered. Error conditions are not described at this level of refinement.

The strategy to be followed for the correctness arguments

It has to be shown that:
A) once a service request is present in a service queue, it remains in the service queue unless the service request is allocated to a service provider simulator (the safety part) and
B) eventually a service request is allocated to a service provider simulator (the liveness part).

The strategy for the informal proof of part A is as follows:
A1) Find all the correctness properties that imply the removal of a service request from a service queue. In this case only one such property is found, viz. the *DP1-01(ServiceCategory)* property.
A2) Show that the service request is assigned to a service provider simulator when it is removed from the service queue.
A3) Since the `postconditions:withArguments:` construct is used to specify the assignment of a service request to a service provider simulator, it needs to be shown that the preconditions of the corresponding method will hold when the latter starts its execution.

A4) The action in A3 is applied recursively until no further `postconditions:` or `postconditions:withArguments:` constructs are found.

The strategy for the informal proof of part B is as follows:
Once the correctness arguments of Part A have been presented, it will be evident that there is only one method which removes a service request from a service queue, viz. the `p_execute` method of the `ServiceCategory` class. It will also be clear that when the service request is removed from the service queue, it is assigned to a service provider simulator (an element of the `spAgentContainer`) in a single atomic action. It now only remains to be shown that the preconditions of the safe liveness correctness property of the `p_execute` method will eventually hold.
B1) The first predicate of the preconditions of this property requires a non-empty `serviceQ`. It therefore needs to be argued that a service request will be present in the `serviceQ`.
B2) The second predicate of the safe liveness property of the `p_execute` method specifies that a service provider simulator will eventually be willing to accept the service request. It therefore needs to be shown that:
B2.1) Each service provider simulator eventually responds with the value true when the `canAcceptNextSR:` message is sent to it, provided the service category passed as parameter matches one of the categories serviced by the service provider simulator.
B2.2) A service category only sends the `canAcceptNextSR:` message to service provider simulators that have indicated their capability to service that particular service category. These service provider simulators are elements of the `spSubset` collection of the `ServiceCategory` instance.
B2.3) The `spSubset` collection of each `ServiceCategory` instance contains at least one element.

Thus, by providing correctness as outlined above, it can be concluded that any service request present in a service queue remains in that queue until it is assigned to a service provider simulator.

The correctness arguments

Part A1:
First of all the classes comprising the call centre system are inspected with the aim of finding a correctness property that implies the removal of a service request from a service queue. The only one that is found, belongs to the `p_execute` method of the `ServiceCategory` class[20]. Its safe liveness property specifies:

```
serviceQ isEmpty not ∧ self canAssignSR ensures
      self postconditions: (#assignToSP:) withArguments:
      #((serviceQ first)) ∧
      serviceQ postconditions: (#removeFirst)
                                        "DP1-01(ServiceCategory)".
```

Part A2:
The next step is to show that a service request is assigned to a service provider simulator when it is removed from the service queue. This is evident from the postconditions of the safe liveness property of the `p_execute` method. These indicate that the `assignToSP:` and `removeFirst` methods are invoked; the former to assign a service request to a service provider simulator and the latter to remove that service request from a service queue. The fact that property *DP1-01(ServiceCategory)* is an **ensures** relation, guarantees that these actions will be executed atomically.

[20] This can be verified by inspecting the statements of the classes in Appendix B. The ServiceCategory class is specified in Appendix B, Section B.10.

Inspection of the correctness properties of the `assignToSP:` method reveals that the `assignToSP:` invokes the `processServiceRequest:` method of the `ServiceProviderSimulator` instance in order to perform the actual assignment of the service request to a service provider simulator, as shown below:

```
"Total correctness property of the assignToSP: method"
sr notNil ∧ availableServiceProvider notNil results-in
      methodReturnValue = self ∧
      sr postconditions: (#serviceProvider:[21])
            withArguments: #(availableServiceProvider) ∧
      availableServiceProvider
            postconditions:(#processServiceRequest:)
            withArguments: #(sr)     "DL1-10 (ServiceCategory)"
```

The `availableServiceProvider` *macro-variable* is defined as:

```
availableServiceProvider ≡
      spSubset detect: [:each | each canAcceptNextSR:
      serviceQCategory]        .
```

It is when the `processServiceRequest:` message is sent to `availableServiceProvider` that the service request is assigned to the service provider, as is evident from the total correctness property of the `processServiceRequest:` method of the `ServiceProviderSimulator` class:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Thus, by inspecting the correctness properties of the classes comprising the call centre, one can deduce that when a service request is removed from a service queue, then it is assigned to a service provider simulator, provided the preconditions of the `assignToSP:` and `processServiceRequest:` methods are met. The next two parts of this informal proof are devoted to showing that the preconditions are indeed satisfied.

Part A3:

Property *DP1-01(ServiceCategory)*, which describes the behaviour of the `p_execute` method, uses the `postconditions:withArgument:` construct to convey the fact that a service request is assigned to a service provider simulator. In order to ensure that the allocation will be successful, it has to be shown that the preconditions of the `assignToSP:` method will always be satisfied when the latter is invoked from within the `p_execute` method.

[21] The serviceProvider: method of the ServiceRequest instance sets the serviceProvider attribute of that instance to the value specified in the argument of the method. In this case it refers to the service provider simulator that will be processing the service request. Refer to Appendix B, Section B.9 for details of this method.

In fact, it can be said that the preconditions of the `p_execute` method of the `ServiceCategory` class were designed with the aim of ensuring that the preconditions of any other methods invoked by the statements of the `p_execute method` would be satisfied vacuously. This is possible because the `p_execute` method contains only one statement that could change the values of the predicates appearing in the precondition of property *DP1-01(ServiceCategory)* and that is the statement that invokes the method(s) that have the same preconditions as the `p_execute` method itself. At this stage it is assumed that the preconditions of the `p_execute` method hold. The correctness arguments to prove that they do indeed hold, are presented in part B. The predicates of the preconditions of the `assignToSP:` method are now considered one by one.

The first predicate of property *DL1-10(ServiceCategory)* of the `p_execute` method (given above) specifies that `sr` should not be nil, where `sr` is the message argument in the message pattern. Thus, `assignToSP:` should be invoked with a non-nil parameter. Upon inspection of property *DP1-01(ServiceCategory)* describing the behaviour of the `p_execute` method, it emerges that the argument that is passed to the `assignToSP:` method is `serviceQ first` (this is clear from the `postconditions:withArguments:` constructs used in that correctness property). The first element of `serviceQ` is guaranteed not to be nil by the `serviceQ isEmpty not` precondition of the *DP1-01(ServiceCategory)* correctness property of the `p_execute` method.

The second precondition of the total correctness property of the `assignToSP:` method specifies that `availableServiceProvider notNil` has to hold. This means that there has to be at least one service provider in `spSubset` that can accept a new service request, as is evident from the definition of the `availableServiceProvider` *macro-variable* listed above.

The second precondition of property *DP1-01(ServiceCategory)* of the `p_execute` method contains the following predicate: `self canAssignSR`. The total correctness property of the `canAssignSR` method of the `ServiceCategory` class ensures that a value of true is only returned if there is at least one service provider simulator in `spSubset` that can accept a new service request, as can be seen below:

```
"Total correctness property of the canAssignSR method"
true results-in methodReturnValue =
      (spSubset detect:
      [:each | each canAcceptNextSR: serviceQCategory]
      ifNone: [nil] ) notNil          "DL1-05(ServiceCategory)"
```

Thus, if the preconditions of the property *DP1-01(ServiceCategory)* are satisfied, then it implies that the `canAssignSR` method returns true. As is evident from the above, this means that the preconditions of the `assignToSP:` method are also satisfied.

Part A4:
As stated earlier, it is not sufficient to check that the preconditions of the methods invoked by the `p_execute` method are satisfied. One also has to ensure that the preconditions of methods invoked by methods invoked by the `p_execute` method are satisifed. Thus, step A3 has to be executed recursively. The `assignToSP:` method, which is called from within the `p_execute` method, in turn also invokes other methods (this is evident from the presence of the `postconditions: withArguments:` constructs in its total correctness property specification given above). One therefore needs to check that the preconditions of those methods are also satisfied in order to ensure that their postconditions will hold.

The precondition of the `serviceProvider:` method of the `ServiceRequest` class[22] is specified as `true` and is therefore satisfied vacuously.

The total correctness property of the `processServiceRequest:` method is as follows:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

The first predicate involves the `categoryIndex` instance variable of the `ServiceProviderSimulator` class. The latter does not export any methods to modify this variable, so it is the responsibility of the `ServiceProviderSimulator` instance to ensure that the value of `categoryIndex` is restricted to the specified range. This is guaranteed by the class invariant shown below:

```
invariant categoryIndex ≥ 0 ∧
      categoryIndex < nrOfCategoriesServed
                          "DS2-01 (ServiceProviderSimulator)"
```

"The categoryIndex is always greater than or equal to zero and less than nrOfCategoriesServed."

The second predicate in the preconditions of the `processServiceRequest:` method specifies that `aServiceRequest` (the argument of the method)[23] should not be nil. Since the `assignToSP:` method invokes the `processServiceRequest:` method passing its *pseudo-variable* `sr` as the argument, and since the preconditions of the `assignToSP:` method in turn requires `sr` to be not nil, the value of `aServiceRequest` is guaranteed not to be nil. (The value of `sr` cannot be changed by the `assignToSP:` method itself, since the value of a *pseudo-variable* may not be changed.)

The third precondition of the total correctness property of the `processServiceRequest:` method requires that a `ServiceProviderSimulator` instance should be willing to accept `aServiceRequest`, i.e. the `canAcceptNextSR:` method has to return the value true. As shown earlier in this section, the `assignToSP:` method is only executed if its preconditions are satisfied. This means the `availableServiceProvider`[24] `notNil` predicate of the preconditions of the `assignToSP:` method will always be true when the `processServiceRequest:` method is invoked, since the latter is invoked from within one of the statements of the `assignToSP:` method and there are no statements[25] in `assignToSP:` that could change the outcome of the `canAcceptNextSR:` method before `processServiceRequest:` is invoked. It therefore follows that the preconditions of the `processServiceRequest:` method will hold when it is invoked from within the `assignToSP:` method.

---

[22] The SLOOP specification of the ServiceRequest class is given in Appendix B, Section B.9.
[23] The processServiceRequest: method is specified in Appendix B, Section B.13, where the usage of its argument is shown.
[24] The availableServiceProvider macro definition was discussed in part A2.
[25] This can be verified by checking the statements of the assignToSP: method in Appendix B, Section B.10.

The above arguments have shown informally that the service request remains in the service queue unless it is assigned to a service provider simulator. This concludes Part A of the correctness arguments.

Part B:
For Part B of the correctness argument it needs to be shown that the postconditions of the *AP1-06(CC_Activation)* safe liveness property will eventually be satisfied, provided the precondition holds. Thus, it has to be shown that there exists a statement which will assign an element of a service queue to a service provider simulator. This is guaranteed by the safe liveness property of the `p_execute` method of the ServiceCategory class. To recapitulate, this property states that:

```
serviceQ isEmpty not ∧ self canAssignSR ensures
        self postconditions: (#assignToSP:) withArguments:
        #((serviceQ first)) ∧
        serviceQ postconditions: (#removeFirst)
                                                "DP1-01(ServiceCategory)".
```

In order to ensure that the postconditions will eventually become true, the preconditions have to be satisfied eventually. This is discussed in parts B1 and B2 of the informal proof.

Part B1:
The first predicate, viz. `serviceQ isEmpty not`, follows directly from the preconditions of property *AP1-06 (CC_Activation)*. Thus it follows directly from the premise of the whole discussion.

Part B2:
The second predicate requires the `canAssignSR` method to return the value `true`. Recall that the total correctness property of the `canAssignSR` method is as follows:

```
true results-in methodReturnValue =
        (spSubset detect:
        [:each | each canAcceptNextSR: serviceQCategory]
        ifNone: [nil]) notNil          "DL1-05(ServiceCategory)"
```

Thus, the method returns true if there is at least one service provider simulator which returns `true` when the `canAcceptNextSR:` message is sent to it.

Part B2.1:
The first step is to show that each service provider simulator will eventually respond with the value `true` when it receives the `canAcceptNextSR:` message, provided that the service category that is passed as parameter is an element of the collection of service categories that it services. This is guaranteed by the liveness property specified for the `ServiceProviderSimulator` class, viz.

```
<∀ categoryIndex where 0 ≤ categoryIndex ∧
categoryIndex < nrOfCategoriesServed ::
¬(self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1)))
leads-to
        self canAcceptNextSR:
        (categoriesServed at: (categoryIndex + 1))
>                          "DL2-01 (ServiceProviderSimulator)"
```

"For any service category serviced by the service provider simulator, the service provider simulator will eventually be able to service a request from that service category."

One now needs to show that each service category only interrogates the service provider simulators that service that particular category. That is the topic of Part B2.2.

Part B2.2:
From the total correctness property of the `canAssignSR` method given at the start of Part B2, it is clear that the `ServiceCategory` instance only invokes the `canAcceptNextSR:` method on members of its `spSubset` collection. This is the collection that contains the service provider simulators that will service requests from the service queue belonging to the `ServiceCategory` instance. This is evident from the following total correctness properties of the `ServiceProviderSimulator` and `ServiceCategory` classes respectively.

Property *DL1-05* of the `ServiceProviderSimulator` class specifies the behaviour of the simulator when it registers itself with the `ServiceCategory` instances that it services. This property specifies, *inter alia*, that the `ServiceProviderSimulator` instance invokes the `addSP:` method of the `ServiceCategory` class for every `ServiceCategory` instance that it services:

```
"Total correctness property of the
registerServiceProvider:using: method"
true results-in
      methodReturnValue = self ∧
      <∀aServiceCategory where
      scContainer includes: aServiceCategory ∧
      aServiceCategory servicedBy: serviceProviderCategory ::
      aServiceCategory postconditions: (#addSP:)
      withArguments: #(self) ∧
      categoriesServed includes:
            (aServiceCategory serviceCategory)
      > ∧
      nrOfCategoriesServed = categoriesServed size ∧
      categoryIndex ≥ 0  "DL1-05 (ServiceProviderSimulator)"
```

In turn, property *DL1-09* of the `ServiceCategory` class specifies that the `addSP:` method adds the service provider simulator passed as parameter (via the `anSP` *pseudo-variable*) to the `spSubset` collection:

```
"Total correctness property of the addSP: method"
anSP notNil results-in
      methodReturnValue = self ∧
      spSubset includes: anSP        "DL1-09 (ServiceCategory)"
```

Thus, the `spSubset` of each `ServiceCategory` instance contains all the service provider simulators that are able to process service requests belonging to that particular service category.

Part B2.3:
It now remains to be shown that the `spSubset` collection of each `ServiceCategory` instance will have at least one element. This follows directly from property *AS2-10(CC-Activation)*, which specifies that the service provider subset of each service category contains at least one element.

AS2-10. The service provider subset of each service category contains at least one (simulated) service provider instance.

This concludes the correctness arguments for part B and thus for property *AP1-06 (CC_Activation)*. In this section the emphasis has been on the `postconditions:` and `postconditions:withArguments:` constructs. It was demonstrated how these constructs **highlight** the fact that **other methods are being invoked** from within the method under discussion. It was also shown what role these constructs play in correctness arguments.

Another aspect worth noting here is the role that correctness arguments play in the discovery of design flaws. In the call centre example the original version of the `canAcceptNextSR:` method returned true if no service request was assigned to the service provider simulator. The method did not take any service categories into account. However, it was while working through the correctness arguments of property *AP1-06* that it became clear that in the original version of the design, starvation of a specific service category was possible.

That could have happened if there were multiple `ServiceCategory` instances and whenever a particular `ServiceCategory` instance executed its `p_execute` method, then the `ServiceProviderSimulators` would be busy with a service request from one of the other categories. Thus, the design flaw was discovered while trying to prove that the `canAcceptNextSR:` method will eventually return true when invoked by a specific `ServiceCategory` instance (i.e. while trying to prove part B2 of the above correctness arguments). As a result the design was modified to ensure that the service categories were serviced in a round robin fashion, unless the associated service queues were empty.

7.3.3.2 Using an absence of unsolicited reponse property to demonstrate how the characteristics of an ensures relation can be used in correctness arguments

In this section the focus is on the significance of the **ensures** relation in correctness arguments. This relation is distinguished from similar relations such as **leads-to** and **until** by the fact that the transition from the state where the preconditions are holding to where the postconditions are holding occurs in a single atomic step. If a correctness property contains an **ensures** relation, one is therefore guaranteed that the precondition will hold up until the point when the postconditions start to hold. This concept is illustrated via the correctness arguments of property *AP2-01 (CC_Activation)*. This property is as follows:

```
<∀ spAgentContainerElement where
spAgentContainer includes: spAgentContainerElement ::
    <∀ aServiceRequest ::
      < ∃ aServiceQueue ::
      aServiceQueue includes: aServiceRequest precedes
      spAgentContainerElement serviceRequest = aServiceRequest
      >
    >
>                                                    "AP2-01 (CC_Activation)"
```

"A service request is assigned to an element of the service provider container only if the service request has been enqueued in a service queue and has remained in the queue until it was assigned to the service provider container element."

The strategy to be followed for the correctness arguments

It has to be shown that :
A) a service request is only allocated to a service provider simulator if the former has been enqueued in a service queue and
B) once a service request is enqueued in a service queue, it remains there until it is allocated to a service provider simulator.

The correctness arguments

Part A:
A service request is allocated to a service provider simulator via the `processServiceRequest:` method of the `ServiceProviderSimulator` class, as is evident from the total correctness property of this method:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Upon inspection of the correctness properties of the classes used in the call centre system, it is found that the `processServiceRequest:` method is invoked only by the `assignToSP:` method of the `ServiceCategory` class. In the total correctness property of the `assignToSP:` method the *pseudo-variable* `sr` refers to the service request that is received as argument of the `assignToSP:` method. In turn, the `assignToSP:` method passes the value of `sr` as argument to the `processServiceRequest:` method.

```
"Total correctness property of the assignToSP: method"
sr notNil ∧ availableServiceProvider notNil results-in
      methodReturnValue = self ∧
      sr postconditions: (#serviceProvider:[26])
            withArguments: #(availableServiceProvider) ∧
      availableServiceProvider
            postconditions:(#processServiceRequest:)
            withArguments: #(sr)     "DL1-10 (ServiceCategory)"
```

The `availableServiceProvider` *macro-variable* is defined as:

```
availableServiceProvider ≡
      spSubset detect: [:each | each canAcceptNextSR:
      serviceQCategory]        .
```

The value of `sr` is determined by the `p_execute` method which invokes the `assignToSP:` method. From the precedence property of the `p_execute` method of the `ServiceCategory` class it is clear that the service request that is assigned to a service provider simulator is taken from a service queue:

```
"Safe liveness property of the p_execute method:"
serviceQ isEmpty not ∧ self canAssignSR ensures
      self postconditions: (#assignToSP:) withArguments:
      #((serviceQ first)) ∧
      serviceQ postconditions: (#removeFirst)
                                    "DP1-01(ServiceCategory)".
```

The fact that property *DP1-01(ServiceCategory)* contains an **ensures** relation, guarantees that the service request is removed from the `serviceQ` and assigned to the service provider simulator in

[26] The serviceProvider: method of the ServiceRequest instance sets the serviceProvider attribute of that instance to the value specified in its argument. In this case it refers to the service provider simulator that will be processing the service request. Refer to Appendix B, Section B.9 for details of this method.

a single atomic step. Thus, a service request that is assigned to a service provider simulator, is always taken from a service queue. This concludes the first part of the correctness argument.

Part B:
The second part of the correctness argument reuses the results of property *AP1-06 (CC_Activation)*. In the section 7.3.3.1 it was shown that:

```
<∀ aServiceRequest ::
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
      >
      ensures
            < ∃ aServiceProviderSimulator where
            spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator serviceRequest = aServiceRequest
            >
>                                                    "AP1-06 (CC_Activation)"
```

"A service request remains in a service queue until it is allocated to an element of the service provider container."

Again the property being reused contains an **ensures** relation, which guarantees that the service request will remain in the service queue until it is assigned to a service provider simulator. Property *AP2-01 (CC_Activation)* follows from parts A and B of the correctness argument.

The example in this section has highlighted how the characteristics of a relation such as **ensures** are used in correctness arguments. When an **ensures** relation appears in a correctness property, the software designer can safely assume that once the preconditions hold, there can be no **interference** which could affect the postconditions specified for the property.

7.3.3.3 Using a fair responsiveness property to illustrate the importance of showing via correctness arguments that the selection of the constituent classes of a system will indeed result in the behaviour as described in the specification of the system

This section illustrates a specific aspect of the role of correctness arguments in the SLOOP method, viz. that the correctness arguments are used to check **informally** that the **actual behaviour** of the system as implied by the **constituent classes** of the SLOOP program matches the **specified behaviour** of the system as implied by the correctness properties of the system. The correctness arguments given below for property *AP3-01 (CC_Activation)* exemplify this aspect of correctness reasoning.

When a service request reaches the head of the `inputQ`, the `ServiceCategoryAllocator`[27] class is used to categorise the service request. This procedure can be quite elaborate, e.g. in the case where a database has to be consulted in order to obtain specific information. It is therefore possible that the `ServiceCategoryAllocator` class could be designed to use parallel statements to perform the categorisation of the service request. Thus, it might not necessarily be an atomic action. One design option is to start the categorisation in a FIFO order, but to allow the service requests to be added to the appropriate service queues as the categorisation for each service request finishes (which might not necessarily correspond to the start order).

Since the parallel statements may be executed in any order, it is possible that even if the service requests are categorised in the order in which they were entered into the `inputQ`, they may still not be assigned to the service queues in that order. However, the requirements analysis states that the service requests should be assigned to the service queues in the order in which the connections were established, which is reflected by property *AP3-01 (CC_Activation)*. When

[27] The ServiceCategoryAllocator class is defined in Appendix B, Section B.8.

selecting the class which has to perform the categorisation of the service requests, it has to be shown that the properties of this class do not violate the properties of the system under development. Property *AP3-01 (CC_Activation)*, which represents one of the correctness properties of the call centre system, is used in the example below to illustrate how the selection of the `ServiceCategoryAllocator` class preserves property *AP3-01(CC_Activation)*.

First of all property *AP3-01(CC_Activation)* is specified more formally. Its specification has two parts: the first part specifies that **if** a service request is added to the `inputQ`, then it is always added to the **end** of the `inputQ` and the second part specifies that **if** `aServiceRequestX` is ahead of `aServiceRequestY` in the `inputQ`, then `aServiceRequestX` **will always** be removed first from the `inputQ`.

```
<∀ aServiceRequestX where
      ¬ (inputQ includes: aServiceRequestX) ::
      ¬ (inputQ includes: aServiceRequestX) unless
      inputQ last = aServiceRequest
 > ∧
<∀ (aServiceRequestX, aServiceRequestY) where
      aServiceRequestX ~~ aServiceRequestY ∧
      inputQ includes: aServiceRequestX ∧
      inputQ includes: aServiceRequestY ::

      inputQ indexOf: aServiceRequestX <
      inputQ indexOf: aServiceRequestY ensures

      ¬ (inputQ includes: aServiceRequestX) ∧
      inputQ includes: aServiceRequestY ∧
      <∃ aServiceQueue :: aServiceQueue includes: aServiceRequestX
      >
>                                              "AP3-01 (CC_Activation)"
```

"Service requests are added and removed from the input queue on a First In First Out basis."

The significance of the **unless** and **ensures** relations in the above property is as follows: The **unless** relation indicates that **if** the service request is added to the `inputQ`, then it will be added to the end of the queue, but it does not guarantee that a service request **will** eventually be added to the `inputQ`. The **unless** relation therefore specifies the behaviour of the system **if** a service request has to be added to the `inputQ` (it describes a **safety** aspect of the system behaviour).

In contrast, the **ensures** relation has both safety and liveness characteristics. The **safety** aspect specifies that **if** a service request is ahead of another service request in the `inputQ` then it will always be processed first. The **liveness** aspect specifies that if a service request is present in the `inputQ`, then it **will eventually** be removed from the `inputQ` and added to a service queue.

Error conditions are not specified at this level of refinement. Service queues are quantified over all service queues associated with service categories in the `scContainer`. This quantification is not shown in order to make the specification less cluttered.

The informal proof of this property is now presented, based on the assumption that the `ServiceCategoryAllocator` is selected as the class to perform the categorisation of the service requests and the allocation of service requests to service queues. By showing the correctness of the above property based on this assumption, it can be deduced that the `ServiceCategoryAllocator` class does not violate the above property.

The strategy to be followed for the correctness arguments

It has to be shown that
A) **if** a service request is added to the `inputQ`, then it is always added to the **end** of the `inputQ`,
B) if `aServiceRequestX` and `aServiceRequestY` are both present in the `inputQ` and `aServiceRequestX` is ahead of `aServiceRequestY`, then `aServiceRequestX` is removed from the `inputQ` and allocated to a service queue **before** `aServiceRequestY` and
C) **eventually** `aServiceRequestX` is allocated to a service queue while `aServiceRequestY` remains in the `inputQ`.

In turn, part B also has three parts. It is shown that
B1) a service request is always removed from the head of the `inputQ`,
B2) the relative ordering between elements of the `inputQ` is maintained and
B3) when a service request is removed from the `inputQ`, it is added to a service queue.

The correctness arguments

Part A:
Inspection of the classes that constitute the call centre system yields the `CommsProviderSimulator`[28] class as the only one containing a method which adds a service request to the `inputQ`. Safe liveness property *AP1-01(CommsProviderSimulator)* of the `p_generateEvent: target:` method specifies the following:

```
generatingEvent ∧ idleConnection notNil ∧
¬(inputQ includes:(idleConnection serviceRequest)) ensures
        ¬generatingEvent ∧ newEventRequired ∧
        inputQ last = (idleConnection serviceRequest) ∧
        idleConnection postconditions: (#assign)
                                "AP1-01 (CommsProviderSimulator)"
```

where

```
idleConnection ≡ self getIdleConnection: userConnections
```

Thus, when a service request is added to the `inputQ`, it is always added at the end. This concludes part A of the informal proof.

Part B:
The next step is to show that if `aServiceRequestX` and `aServiceRequestY` are both present in the `inputQ` and `aServiceRequestX` is ahead of `aServiceRequestY`, then `aServiceRequestX` is removed from the `inputQ` and allocated to a service queue **before** `aServiceRequestY`.

Part B1:
It must first be shown that a service request is always removed from the head of the `inputQ`. Another inspection of the call centre classes reveals that the only method that results in the removal of a service request from the `inputQ`, is the `p_allocate:from:` method of the `ServiceCategoryAllocator`[29] class. The relevant safe liveness property is shown below.

[28] The CommsProviderSimulator class is specified in Appendix B, Section B.6.
[29] The ServiceCategoryAllocator class is defined in Appendix B, Section B.8.

```
<∀ aServiceRequest where
¬(inputQ isEmpty) ∧ inputQ first == aServiceRequest ::
      aServiceRequest serviceRequestCategory notNil ∧
      aServiceRequest serviceQ isNil ∧
      < ∃ aServiceCategory where
      scContainer includes: aServiceCategory ::
            aServiceCategory serviceQCategory =
            aServiceRequest serviceRequestCategory
      >
 ensures
      self postconditions: (#assignToSQ:using:)
      withArguments: #(aServiceRequest scContainer) ∧
      ¬categorising ∧
      ¬(inputQ includes: aServiceRequest)
>                                    "DP1-04(ServiceCategoryAllocator)"
```

From the above property it is clear that the service request being dealt with here is the one at the head of the `inputQ`. (The variable `aServiceRequest` is defined as being equivalent to `inputQ first`.) The postconditions of property *DP1-04(ServiceCategoryAllocator)* indicate that `aServiceRequest` is no longer an element of `inputQ` after the `p_allocate:from:` method has completed its execution. Since this is the only method that results in the removal of a service request from the `inputQ`, it means that service requests are always removed from the head of the `inputQ`, which concludes part B1 of the correctness arguments.

Part B2:
The `inputQ` is created as an instance of the Smalltalk `OrderedCollection` library class. One of the properties of that class is that it maintains the relative ordering of its elements. There are also no statements in the `CallCentreSimulation` program that add or remove elements from the `inputQ` other than those described in parts A and B1. This means that if `aServiceRequestX` is ahead of `aServiceRequestY` in the `inputQ`, then `aServiceRequestX` will always be removed from the `inputQ` before `aServiceRequestY`.

PartB3:
It now remains to be shown that when a service request is removed from the `inputQ`, then it is added to a service queue. In property *DP1-04(ServiceCategoryAllocator)*, which was presented in part B1, it is stated that when `aServiceRequest` is removed from the `inputQ`, then the `assignToSQ:using:` method of the `ServiceCategoryAllocator` class is also executed by the same statement (property *DP1-04(ServiceCategoryAllocator)* is an **ensures** relation).

The `assignToSQ:using:` method is invoked using `aServiceRequest` as one of its arguments. One therefore has to check whether the execution of this method results in the allocation of `aServiceRequest` to a service queue. This is indeed the case, as is evident from the total correctness property of the `assignToSQ:using:` method of the `ServiceCategoryAllocator` class (the *pseudo-variable* `serviceRequest` used in the `assignToSQ:using:` method corresponds to the *pseudo-variable* `aServiceRequest` passed as an argument to the `assignToSQ:using:` method):

```
      "Total correctness property of the assignToSQ:using: method"
      serviceRequest serviceQ isNil ∧
      serviceRequest serviceRequestCategory notNil ∧
      match notNil results-in
            methodReturnValue = self ∧
            serviceRequest serviceQ notNil ∧
            serviceRequest serviceRequestCategory notNil ∧
            (match serviceQ) includes: serviceRequest
                                    "DL1-04(ServiceCategoryAllocator)"
```

The *macro-variable* `match` is defined as:

```
match ≡ scContainer detect: [:each | each serviceQCategory =
      serviceRequest serviceRequestCategory] ifNone: [nil]
```

The preconditions of the `assignToSQ:using:` method are also preconditions of the `p_allocate:from:` method. Since there is only one statement in the `p_allocate:from:` method and that is the one invoking the `assignToSQ:using:` method, these preconditions will therefore still hold when the `assignToSQ:to:` method is invoked from within the `p_allocate:from:` method. The statements of the `p_allocate:from:` method are shown below for easy reference.

```
"The statements of the p_allocate:from: method"
parallel
self assignToSQ: serviceRequest using: scContainer \+
categorising := false \+
inputQ removeFirst
      if serviceRequest notNil and:
      [serviceRequest serviceRequestCategory notNil and:
      [serviceRequest serviceQ isNil]]
end-parallel
```

Thus, if the preconditions of the `p_allocate:from:` method hold when the latter is executed, then the service request at the head of the `inputQ` will be removed and the `assignToSQ:using:` method will be executed. As shown above, the preconditions of the `assignToSQ:using:` method will hold when this method is invoked, therefore the postconditions of the `assignToSQ:using:` method are guaranteed to hold after its execution (i.e. the service request will have been added to a service queue).

Thus, since

- a service request is always removed from the head of the `inputQ`,
- the relative ordering of the elements of the `inputQ` is always maintained and
- the removal of a service request from the `inputQ` coincides with its allocation to a service queue,

the following is implied:

If the index of `aServiceRequestX` in the `inputQ` is less than the index of `aServiceRequestY` in the `inputQ`, then `aServiceRequestX` is allocated to a service queue before `aServiceRequestY`. This concludes part B of the correctness arguments.

Part C:

Part C of the correctness argument reuses the results of property *AP1-05 (CC_Activation)*.

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest ensures
      ¬(inputQ includes: aServiceRequest) ∧
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest)
      >
>                                                "AP1-05 (CC_Activation)"
```

"A service request remains in the inputQ until it is assigned to a service queue."

The above property implies that each service request **will eventually** be removed from the `inputQ`. If the index of `aServiceRequestX` is less than that of the index of `aServiceRequestY`, then `aServiceRequestX` will eventually be removed from the `inputQ`, while `aServiceRequestY` will still be an element of `inputQ`. This is because the relative ordering of the elements of the `inputQ` always remains the same (as was shown in part

B2) and service requests are always removed from the head of the `inputQ` (as was demonstrated in Part B1). This concludes the correctness argument for part C and thus also for property *AP3-01 (CC_Activation)*.

By presenting the above correctness arguments based on the assumption that the `ServiceCategoryAllocator` class is used to perform the categorisation of the service requests and the allocation of these requests to service queues, it is implied that this class does not violate this property. The correctness properties specified for the system under development are therefore used not only at the **beginning** of the design phase in order to **aid the selection** of the right constituent classes of the system, but also at the **end** of the design phase to **confirm** the correctness of their selection.

7.4 Deriving SLOOP statements from correctness properties

It is now shown how correctness properties can be used to derive SLOOP statements. The example that was introduced very briefly in Chapter 4, Section 4.2.3, is used to demonstrate the concepts.

The functionality of the required system is as follows: A dispatcher system must be developed which acts as a generic transformer of objects, receiving them from a producer, performing some transformation on the received objects and then dispatching them to a consumer. The dispatcher has to queue the objects in a FIFO order until the consumer is ready to receive the next object. The dispatcher keeps a record of the maximum length ever reached by the queue managed by the dispatcher. The dispatcher ensures that the size of this queue never exceeds a specified maximum value.

The following Dispatcher class attributes can be identified:

`bufferedElements`	This attribute refers to the FIFO queue managed by the Dispatcher class.
`consumer`	This attribute refers to the instance of the Consumer class to which the Dispatcher instance will dispatch the elements in its `bufferedElements` queue.
`maximumRecordedLength`	This attribute represents the maximum length ever reached by the `bufferedElements` queue.
`maximumAllowedLength`	This attribute represents the maximum length that the `bufferedElements` queue may ever reach.
`newElement`	This attribute refers to an object which has been received from the Producer instance, but which has not yet been added to the `bufferedElements` queue.

The correctness properties below specify the required behaviour of the Dispatcher class:

invariant `newElement notNil ⇒ ¬ self readyToReceiveElement`
"AS2-01 (Dispatcher)"

"The dispatcher will never indicate that it is ready to accept a new element from the producer if it has not yet added the element passed to it on a previous occasion to the bufferedElements queue."

invariant `bufferedElements size <= maximumAllowedLength`
"AS2-02 (Dispatcher)"

"The current length of the bufferedElements queue is always less than or equal to the maximumAllowedLength."

```
invariant maximumRecordedLength >= bufferedElements size
                                                    "AS3-01 (Dispatcher)"
```

"The maximumRecordedLength of the bufferedElements queue is always greater than or equal to the current queue size."

```
< ∀ k where 0 <= k ∧ k <= maximumAllowedLength ::
maximumRecordedLength = k unless maximumRecordedLength > k
>                                                   "AS4-01 (Dispatcher)"
```

"The maximumRecordedLength of the bufferedElements queue is non-decreasing."

```
< ∀ anElement where newElement = anElement ::
newElement = anElement unless
      self postconditions: (#transform:) withArguments: #(anElement) ∧
      bufferedElements includes: anElement ∧ newElement isNil
>                                                   "AS4-02 (Dispatcher)"
```

"Once the dispatcher has accepted a new object from the producer, it remains a new object unless it is transformed and added to the bufferedElements queue."

```
< ∀ anElement where bufferedElements includes: anElement ::
bufferedElements includes: anElement unless
      consumer postconditions: (#pass:)  withArguments: #(anElement)
>                                                   "AS4-03 (Dispatcher)"
```

"Once an object is added to the bufferedElements queue, it remains there unless it is passed to the consumer."

```
<∀ anElementY) where
      ¬ bufferedElements includes: anElementY ::
      ¬ bufferedElements includes: anElementY unless
      bufferedElements last = anElementY
 >                                                  "AS4-04 (Dispatcher)"
```

"If an object is added to the bufferedElements queue, it is always added to the end of the queue."

```
<∀ (anElementX, anElementY) where
      anElementX ~~ anElementY ∧
      bufferedElements includes: anElementX ∧
      bufferedElements includes: anElementY ::

      bufferedElements indexOf: anElementX <
      bufferedElements indexOf: anElementY unless

      ¬ bufferedElements includes: anElementX) ∧
      bufferedElements includes: anElementY
>                                                   "AS4-05 (Dispatcher)"
```

"Objects are removed from the bufferedElements queue in the order that they are added to the queue."

The precedence properties are as follows:

```
< ∀ anElement where newElement = anElement ::
newElement = anElement ∧
bufferedElements size < maximumAllowedLength ensures
      self postconditions: (#transform:) withArguments: #(anElement) ∧
      bufferedElements includes: anElement ∧ newElement isNil
>                                                   "AP1-01 (Dispatcher)"
```

"If the dispatcher has received a new object from the producer and the size of the bufferedElements queue is less than its maximumAllowedLength, these conditions continue to hold until the new object is transformed and added to the bufferedElements queue."

```
< ∀ anElement where bufferedElements includes: anElement ::
bufferedElements includes: anElement ∧
consumer readyToReceiveElement ensures
        consumer postconditions: (#pass:) withArguments: #(anElement) ∧
        ¬ bufferedElements includes: anElement
>                                                    "AP1-02 (Dispatcher)"
```

"Once the consumer is ready to receive an object and the bufferedElements queue is non-empty, these conditions continue to hold until an element of the bufferedElements queue is removed from the queue and passed to the consumer."

The methods of the various classes of a system are defined once the design phase correctness properties have been specified. The behaviour described by the correctness properties yields the necessary information to derive the methods of the classes. The liveness and/or precedence properties provide the basis for deriving the parallel methods. The infinitely often execution of the statements of the parallel methods of the various classes has to result in the desired progress being made.

In the Dispatcher class example, properties *AP1-01* and *AP1-02* specify the progress that needs to be made by the Dispatcher class instance. Briefly, if the latter has accepted a new element from the Producer instance, then the new element should be transformed and added to the `bufferedElements` queue once there is space in that queue. In addition, the elements in the `bufferedElements` queue should be passed to the Consumer instance whenever the latter is ready to accept them. The above describes the crux of the functionality of the Dispatcher class. One or more parallel methods can be defined to contain the parallel statements that realise this functionality. In the case of the Dispatcher class a single parallel method called `p_dispatch` suffices, because the functionality of this class is very simple.

A useful heuristic for deriving parallel statements from properties of the form "*p* **ensures** *q*", is to populate the *if* clause using the information in the conjuncts in *p* and to use the conjuncts in *q* to determine the state changes. Note however that the conjuncts in the correctness property are not mapped to expressions in the SLOOP statement in a mechanical way. For example, in property *AP1-01* universal quantification is used and there are therefore references to `anElement`. The introduction of the variable `anElement` is necessary in order to be able to refer to the old value of the variable `newElement` (i.e. prior to the execution of the statement) in predicate *q*. Due to the evaluation order of a SLOOP parallel statement (discussed in Chapter 4, Section 4.3.6.3), it is not necessary to have such a variable in the SLOOP statement.

Thus, the aim is to devise a statement which, if executed infinitely often, will satisfy correctness property *AP1-01*. We therefore have the following statement in the `p_dispatch` method:

```
self transform: newElement \+
bufferedElements add: newElement \+
newElement := nil
    if newElement notNil and:
    [bufferedElements size < maximumAllowedLength]    "Statement S1"
```

From property *AP1-02* we have the following:

```
consumer pass: anElement \+
bufferedElements remove: anElement
```

```
    if consumer readyToReceiveElement and:
    [bufferedElements includes: anElement]          "Statement S2"
```

At this stage statement *S2* still refers to `anElement`. Property *AP1-02* does not provide any additional information regarding the identity of `anElement`. It will be necessary to consider the other correctness properties before statement *S2* can be refined further.

The derivation of the above two statements demonstrates the advantages of the SLOOP computational model. These statements are designed without having to be concerned about location counters. For example, when specifying statement *S2*, there is no need to consider the location counter of the consumer object. The only important issue is to identify the conditions that need to be satisfied in order for statement *S2* to produce the desired result. Since each parallel statement is executed infinitely often, the correct results will be achieved, provided its preconditions are true at some point and remain true until the statement has executed.

The above statements take care of the liveness and precedence properties. However, the safety properties have to be considered as well. They provide the necessary information for the refinements of these statements.

Property *AS4-04* prescribes how the new object should be added to `bufferedElements`, i.e. always at the end. Consequently the message to `bufferedElements` in statement *S1* is modified from

```
bufferedElements add: newElement
```

to

```
bufferedElements addLast: newElement
```

The resulting statement is as follows:

```
   self transform: newElement \+
   bufferedElements addLast: newElement \+
   newElement := nil
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength]   "Statement S1"
```

Property *AS3-01* (**invariant** `maximumRecordedLength >= bufferedElements size`) describes the invariant relationship between the size of `bufferedElements` and the value of `maximumRecordedLength`. When inspecting statement *S1* to determine whether its execution could ever violate this invariant, it is found that such a possibility exists with the execution of the statement component `bufferedElements addLast: newElement`. If the latter is executed, the size of `bufferedElements` is implicitly incremented by one. One therefore has to ensure that the value of `maximumRecordedLength` is updated whenever both of the following conditions are true: (1) the size of `bufferedElements` is incremented and (2) the new size will be greater than the current value of `maximumRecordedLength`.

This can be achieved by adding an additional component to statement *S1*. By making it part of the same statement, one ensures that the components will be executed as one atomic action. This yields a new statement *S1*:

```
   self transform: newElement \+
   bufferedElements addLast: newElement \+
   newElement := nil
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength]


|| maximumRecordedLength := bufferedElements size + 1
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength and:
```

```
    [bufferedElements size +1 > maximumRecordedLength] ]
                                                    "Statement S1"
```

Recall that all *if* clauses of all components of a particular statement are evaluated before any of the component parts are executed. All evaluations of `bufferedElements size` in statement *S1* will therefore yield the same result. Thereafter all the message expressions as listed in step 2 in Chapter 4, Section 4.3.6.3, are evaluated, followed by the evaluation of all message expressions and assignments as described in step 3 in Section 4.3.6.3.

As part of step 2 the following values are obtained (in any arbitrary order):
`self, newElement, bufferedElements, nil, (bufferedElements size + 1).`

As part of step 3 the following assignments and message expressions are executed (in any arbitrary order):

```
self transform: newElement
bufferedElements addLast: newElement
newElement := nil
maximumRecordedLength := bufferedElements size + 1
```

Since `newElement` was evaluated in step 2, the assignment of the value nil to the variable `newElement` in the third *component-part* does not affect the first or second *component-parts*. Similarly, because the value of `bufferedElements size + 1` was determined in step 2, the assignment to `maximumRecordedLength` is not affected by the execution of the `bufferedElements addLast: newElement` *component-part*.

Property *AS4-05* specifies that elements should only be removed from the head of `bufferedElements`, which provides us with the necessary information to replace `anElement` with a more specific description in statement *S2*. All references to `anElement` are therefore changed to `bufferedElements first`.

```
consumer pass: (bufferedElements first) \+
bufferedElements removeFirst
   if consumer readyToReceiveElement and:
   [bufferedElements size > 0]                  "Statement S2"
```

The correctness properties also yield a number of sequential methods. Property *AS2-01* results in the definition of sequential method `readyToReceiveElement`. Since each sequential method has to terminate, a total correctness property is defined for each sequential method. The `readyToReceiveElement` method has the following total correctness property resulting from property *AS2-01*:

```
true results-in methodReturnValue = (newElement isNil)
                                                "DL1-01 (Dispatcher)"
```

The need for the `transform:` sequential method is derived from properties *AS4-02* and *AP1-01*. These correctness properties do not specify the behaviour of the `transform:` method. In the Dispatcher class this method is defined to merely return the value of the receiver. In subclasses, various transformations may be defined.

The other sequential methods that are referenced in the correctness properties of the Dispatcher class belong to the OrderedCollection class and to the class of the consumer object. The statements contained in these methods are encapsulated within those classes and are therefore irrelevant to the discussion of the statements of the Dispatcher methods. Only the correctness properties of those methods are important when discussing the Dispatcher class.

While discussing the derivation of the methods of the Dispatcher class, no mention was made of properties *AS2-02*, *AS4-01*, *AS4-02* and *AS4-03*. These correctness properties specify constraints on the methods and statements of the Dispatcher class. Rather than indicating what statements should be included, they dictate what may not be included. For example, property *AS2-02* ensures that there will be no statement that will cause the size of the `bufferedElements` queue to exceed `maximumAllowedLength`.

Although property *AS3-01* ensures that `maximumRecordedLength` is always greater than or equal to the current size of the `bufferedElements` queue, it does not ensure that it is greater than or equal to any **previous** size of the `bufferedElements` queue. This additional constraint is provided by property *AS4-01*, which ensures that no statement will ever set the value of `maximumRecordedLength` to a value less than its current value.

Property *AS4-02* implies that the only time when `newElement` may be set to nil is when the object that it references is added to the `bufferedElements` queue. Consequently the Dispatcher class offers no method which would enable another object to set `newElement` to nil. Similarly, property *AS4-03* implies that the only time when an element is removed from the `bufferedElements` queue is when it is passed to the consumer.

The way in which correctness properties are used in the SLOOP method can be summarised as follows: At the start of the design phase, the repository of reusable artifacts is searched for classes that will match the requirements as outlined by the correctness properties identified during the requirements analysis phase. If new classes have to be designed, then once again these correctness properties provide the necessary information regarding the requirements of these classes. The SLOOP statements of the new classes are derived from the specified correctness properties. Once all the classes have been finalised, informal proofs of the correctness properties specified for the system under development confirm that the selected classes are indeed the correct ones.

7.5 Summary

This chapter has illustrated how the semantics of a class and its methods are conveyed by their correctness properties. It has also demonstrated how the various types of correctness properties can be reasoned about and how SLOOP statements can be derived from correctness properties. Correctness arguments were given for an example property of each type. These correctness arguments were based on other properties that were specified for the system or for the constituent classes, as well as on the SLOOP statements that realised those properties. Each example also highlighted at least one additional aspect of correctness reasoning in the SLOOP method. They were as follows:

- One of the **advantages of using macros** is the following: If a *macro-variable* is defined as the value returned by a method of another class, then the correctness properties regarding that value as defined by the target class may be reused by the client class. If the value had been assigned to an instance variable of the client class, then the latter would have had to specify its own correctness properties regarding the value of the instance variable.
- **Location counters are not considered** during correctness reasoning.
- It is also **not necessary** to be concerned about the **allocation of statements to processors**. At the design level, correctness reasoning is in terms of the **parallel statements**, each of which executes atomically.
- The role of the **computational model** in the correctness arguments is extremely important. There is **no need to consider all possible sequences of events.** It is only necessary to follow the **invocation paths** starting from each parallel statement that is activated via the *activation-section*.

- The **repeated execution of the parallel statements** selected for the program **eventually results in the postconditions** specified by the **liveness** properties, provided their preconditions hold at some point.
- The role of **preconditions** is to define the **responsibility of the client**. When showing that a specific property associated with a method is correct, the preconditions can be assumed to hold. It is merely necessary to show that the postconditions will be achieved, provided the preconditions hold. However, when that correctness property is being **reused** (for example when the corresponding method is being invoked by a client), then it is necessary to show that the preconditions will indeed hold at the time when the method is invoked. In the case where the property is used to ensure **progress**, it is necessary to show that the **preconditions will eventually hold**.
- The important aspect regarding reasoning about **total correctness** properties of sequential methods is the fact that **concurrency** does not need to be taken into account. **Statement interleaving** takes place at the level of **parallel** statements, which may invoke sequential methods. Each parallel statement executes **atomically**.
- Since a sequential statement executes as a single atomic unit, the postconditions of a total correctness property are always achieved during the execution of a **single** parallel statement. This means that the **ensures** logical relation (which requires its postconditions to be achieved via a single parallel statement) can include invocations of sequential methods.
- The distinctive properties of the **leads-to** logical relation, such as its transitive properties, can be utilised in correctness arguments.
- The **reuse** of correctness properties has several benefits. A great deal of **effort is saved** when the properties of classes can be assumed to hold **without having to reason about them from first principles each time**.
- Another benefit results from the usage of the `postconditions:` and `postconditions: withArguments:` constructs. The latter **highlights** the fact that other methods are being invoked by the current method. The usage of such constructs also allows one to **reason about the pre- and postconditions** of the methods being invoked **without** having to **repeat** those conditions in the present correctness property.
- While reusing correctness properties in correctness arguments, one should take care that there is **no circular reasoning** (i.e. in order to prove property A, property B is reused, but property B depends on the correctness of property A).
- **Data encapsulation** ensures that in those cases where the class does not provide any methods to modify a specific class or instance variable, only the correctness properties of the class itself need to be considered when reasoning about the possible values of such a variable. If the class does provide methods to modify the variable, then the pre- and postconditions of those methods must be taken into account and the designer has to ensure that the clients do not violate the preconditions when they invoke the methods. Data encapsulation therefore **restricts the number of correctness properties** that need to be considered during correctness arguments.
- The implications of using **inheritance** are manifold. Correctness properties of ancestor classes can be **reused as is** in the correctness arguments of properties of descendant classes. They can also be **overridden**, provided the **preconditions** are **not strengthened** and the **postconditions** are **not weakened**. New correctness properties may also be **added** in the descendant classes.
- Finally, it should not be underestimated how important it is to use correctness arguments to check **informally** that the **actual behaviour** of the system, as implied by the correctness properties of the **constituent** classes of the SLOOP program, matches the **intended behaviour** of the system, as implied by the specified correctness properties of the system. Correctness arguments are used to **confirm** the **choice of classes** selected to comprise the system.

As was stated earlier, the correctness reasoning during the design phase takes place at the level of the parallel statement, i.e. it is based on the atomicity of the parallel statement. The allocation of these statements to processors is not considered. The next chapter deals with the issue of

ensuring that the semantics of the SLOOP parallel statements are preserved when the statements are allocated to one or more processors, thereby ensuring that the correctness arguments used during the design phase are not invalidated by the mapping procedure.

CHAPTER 8

THE IMPLEMENTATION PHASE

8.1 Introduction

Up until now the target architecture of the system has been ignored, i.e. a unified approach is followed during the analysis and design phases. The implementation phase requires the consideration of a number of issues:

- The target architecture has to be determined.
- The objects and statements have to be assigned to processes/processors.
- The SLOOP program has to be mapped to an executable program.

The target architectures that are discussed in this chapter are:

- a sequential (von Neumann) architecture,
- a synchronous shared-memory architecture,
- an asynchronous shared-memory architecture and
- a distributed system.

The above list is not exhaustive, but was considered sufficiently distinct to demonstrate various types of mappings. These architectures are also discussed in [ChMi88] and it is therefore interesting to compare the mappings described in [ChMi88] with the mappings performed in the SLOOP method.

In Section 4.4.3 of Chapter 4 an overview of the mapping of a SLOOP program to an executable Smalltalk program was presented. It merely served as an introduction to the topic. This chapter elaborates on the following issues:

- Each type of target architecture requires a different type of mapping. The heuristics for the allocation of objects and SLOOP statements to processes / processors are given in Section 8.2.
- In Chapter 4 the derivation of an executable Smalltalk program for a sequential architecture was described. In Section 8.3 it is shown how the Smalltalk program can be adapted for other architectures, such as synchronous shared-memory, asynchronous shared-memory and distributed architectures.
- Further options regarding the mapping of the *macros-section* are discussed in Section 8.4.
- The mapping of more advanced types of SLOOP statements is described in Section 8.5. These descriptions cover the mappings of programs that contain multiple *quantified-statement-lists* as well as statements that contain multiple *statement-components* and *component-parts*.

- In Section 8.6 is shown how the reflective[1] facilities of Smalltalk can be used in order to make the mapping as transparent as possible to the class. Reflective computation can also be used to perform assertion checking.
- When the SLOOP program is mapped to a target architecture, it may be found that the level of parallelism displayed by the SLOOP program is not sufficient. This results in a return to the design phase. The SLOOP program is refined to introduce more parallelism, typically by decoupling the actions that appear in a single statement and by putting them into additional parallel statements. This topic is covered in Section 8.7.

At all times during the implementation phase the correctness properties specified during the analysis and design phases play an extremely important role. Additional correctness properties are defined for the infrastructures used during the implementation phase.

Note that the emphasis in this chapter is on Smalltalk as the target programming language. Where concurrency constructs are not required, the Smalltalk-80 language [GoRo89] suffices, otherwise the Concurrent Smalltalk language [Yoko90] is appropriate. However, the mappings discussed in this chapter are examples only. Many other mappings are possible, including mappings to other programming languages such as Java. As mentioned in Chapter 1, Smalltalk to Java translation has already been studied by other researchers [Enko98].

8.2 Mappings to various architectures

In order to map a SLOOP program to a target architecture, the SLOOP statements have to be **allocated to the process(es) / processor(s)** involved. The following four subsections discuss this allocation in the context of sequential, synchronous shared-memory, asynchronous shared-memory and distributed architectures respectively. Section 8.3 deals with the issues that need to be considered in order to ensure that the **semantics** of the SLOOP statements are retained during the mapping.

8.2.1 Sequential architectures

In the case of a sequential architecture, there is a single processor and a single process. Instructions are therefore executed strictly sequentially [Tane81]. When mapping a SLOOP program to such an architecture, all the statements are assigned to the same process on the same processor. Although the parallel statements are executed sequentially, their order of appearance is irrelevant. The only important issue is that they should be enclosed in an infinite loop in order to ensure that they are executed infinitely often. Each parallel statement should appear at least once within this loop.

8.2.2 Synchronous shared-memory architectures

Synchronous shared-memory architectures allow for multiple processors to share a common memory. There is a common clock and at each clock tick, each processor performs a single step of computation [ChMi88]. Multiple processors may read from the same memory location concurrently. If multiple processors write to the same location concurrently, they all have to write the same value. Concurrent read and write accesses to the same location are not allowed.

This type of architecture is particularly suited to take advantage of the **synchrony** inherent in SLOOP statements. Recall that the *component-parts* of a SLOOP statement (i.e. those parts separated by the '||' or '\+' symbols), execute in parallel. This means that each ***component-part*** of a specific SLOOP parallel statement can be assigned to a **separate processor**.

[1] The concept of computational reflection was described briefly in Chapter 1, Section 1.3.4. Further details are given in Section 8.6.

One statement is executed at a time, with each processor executing a *component-part* of that statement. Infinite loops are implemented on each processor in order to ensure that the *component-part*s of each statement will be executed inifinitely often.

8.2.3 Asynchronous shared-memory architectures

Asynchronous shared-memory architectures also allow for multiple processors to share a common memory. **Asynchronous** shared-memory architectures do **not** have a **common clock**, so the computation steps of the various processors might or might not execute simultaneously. If two processors access the same memory location simultaneously, the actual accesses occur in an arbitrary order [ChMi88].

Each parallel **statement** is assigned to **one** of the processors. (Multiple statements may be assigned to each processor.) If two statements do not send messages to the same object, they may execute concurrently, otherwise they have to execute in an arbitrary sequential order. This is to prevent **interference**[2].

In the case where a statement refers to shared objects, the sequential ordering is achieved in the following way: Before any statement may be executed, all the shared objects that are accessed by that statement have to be reserved (locked) by the processor to which the statement has been allocated. Since each statement may refer to **multiple** objects in the common address space, it would be possible to have a scenario where processor A has been granted a lock on object X and is waiting for a lock on object Y, while processor B has been granted a lock on object Y and is waiting for a lock on object X. Each processor will wait forever for the other to release the required lock. The two processors are therefore in a **deadlock**[3] situation.

One algorithm that guarantees the absence of deadlocks is to reserve the objects in a **prescribed order**. Issues such as performance also need to be considered when selecting an appropriate algorithm. This aspect will be discussed further in the next section, where mappings to distributed systems are described, since the same issues need to be considered when dealing with distributed systems.

In Section 8.3.3 a mapping to a specific type of asynchronous shared-memory architecture is discussed. It describes an architecture that consists of a **single processor**, but which has **multiple processes** running on it. The processes share a common address space. In that case each parallel statement is allocated to a specific process. In the mapping described in Section 8.3.3, no object reservation is required since there is only one processor, but it has to be guaranteed that each statement executes to completion before any other statement can start executing. That is to prevent interference. Although there is only a single processor, the execution of the statements allocated to the various processes can be interleaved in any arbitrary way, so in that sense the program fragments execute in parallel. One can therefore consider it an example of **pseudo-parallelism**.

In order to ensure that all the parallel statements are executed infinitely often when a SLOOP program is mapped to an asynchronous shared-memory architecture, each statement on each processor/process has to execute infinitely often. Thus, all parallel statements allocated to a processor/process have to be enclosed within an infinite loop on that processor/process.

---

[2] The concept of interference was defined in Chapter 2, Section 2.3.2.
[3] The conditions for deadlock were described in Chapter 4, Section 4.3.6.5.

8.2.4 Distributed systems

Distributed systems have multiple processors, each with its own local memory. Communication occurs via **message passing** [Bena90].

When a SLOOP program is mapped to a distributed architecture, each object referenced by the program is allocated to one of the processors. Multiple objects may be allocated to a single processor. Since the parallel and sequential methods of an object are executed on the processor where the object resides, messages have to be passed via some or other communication mechanism if a client invokes a method of a target object that is not co-located. The interface to the communication infrastructure will be discussed in more detail shortly.

At each processor all the parallel statements assigned to a specific process on that processor have to be enclosed in an infinite loop. This ensures that all the parallel statements of the program are executed infinitely often. One of the infinite loops at each processor also has to include a parallel statement that handles the messages received from remote objects. The purpose of this statement is to pass the messages to the relevant local objects.

All the shared objects that are referenced by a parallel statement (either directly from within the parallel statement itself, or indirectly via one of the methods invoked by the parallel statement), have to be **reserved** before the statement may execute. In Section 8.3.4.1 an algorithm for the reservation of resources in a distributed architecture is described. That particular algorithm was chosen for its **simplicity** rather than its **performance**. The objective in this chapter is to point out the issues that are involved, in which case a simple algorithm is the most appropriate.

When an object sends a message to another object and the latter is not co-located, a complex infrastructure is required in order to get the message to the target object. For example, the location of the target object has to be established and the message needs to be converted into a format that can be transmitted over a communication medium. Furthermore, the interface to the communication medium has to be handled. An infrastructure which provides such services is called a **middleware** infrastructure [OHE97]. One example is the Common Object Request Broker Architecture (CORBA) [OHE97].

In this chapter the emphasis is on how to ensure that the **semantics** of the statements of a SLOOP program are **retained** when the program is mapped to a distributed architecture. Once that mapping is done, any middleware infrastructure can be used to take care of the details of actually getting a message across to the target object, provided the selected infrastructure guarantees the reliable delivery of messages to the target object. Thus, the transfer of messages from one processor to another is transparent to the statements of the **mapped** SLOOP program.

8.2.5 Comparison with UNITY mappings

When comparing the UNITY mappings described in [ChMi88] with the SLOOP mappings discussed above, the most important difference lies in the fact that UNITY statements deal with **variables** (all UNITY statements are simple **multiple-assignment** statements), whereas SLOOP statements deal with **objects** and the **messages** sent to those objects. The semantics of a SLOOP statement are therefore more complex and that has to be taken into account when mapping a SLOOP program to a specific architecture. The next section describes in more detail the issues that need to be considered in order to ensure that the semantics of the SLOOP statements are retained during the mapping procedures. Although the object-oriented constructs add complexity to the SLOOP mapping procedures, this is offset by the **higher level of abstraction** that is achieved via the use of object-oriented concepts.

The object-oriented nature of the SLOOP method also gives it a distinct edge over UNITY when the program is mapped to a **distributed** architecture, since the SLOOP mapping can take advantage of middleware infrastructures such as CORBA. The UNITY mappings to distributed architectures are in terms of variables and are described in [ChMi88].

8.3 Deriving executable programs on various architectures

This section demonstrates how executable programs can be derived for the different architectures described in the previous section. The aim is to show what needs to be taken into account in order to ensure that the correctness properties of the system are not violated during the implementation phase.

8.3.1 Sequential architectures

The simplest type of mapping is to a sequential architecture. In Chapter 4, Section 4.4.3, it was shown how an executable Smalltalk program can be derived from a SLOOP program if the target architecture is sequential. Those parts of the executable program that differ for the various architectures are repeated here for the sake of convenience. In the sections that follow, the differences are highlighted.

The mapping of the *activation-section* of the `CallCentreSimulation` SLOOP program is shown below:

```
| aCC_SimulationActivation |

aCC_SimulationActivation :=
      CC_SimulationActivation setup.
[true] whileTrue: [aCC_SimulationActivation p_activate]
```

Thus, an instance of `CC_SimulationActivation` is created. In turn, it instantiates all the necessary classes. The program then enters an infinite loop. The latter contains a single statement which invokes the `p_activate` method of the `CC_SimulationActivation` class. At each invocation of the `p_activate` method one of its constituent parallel statements is executed.

The mapping of the `p_activate` method is now shown. In order to select only one statement at each invocation, while at the same time ensuring that each statement will be selected in turn, two additional instance variables are introduced to the class. The `p_activateTally` variable is set to the number of parallel statements that appear in the method and `p_activateCycleIndex` is initialised to `p_activateTally - 1`. At each invocation of `p_activate`, the `p_activate=CycleIndex` is incremented modulo the number of parallel statements in the method. Its value is then used to select the parallel statement to be executed.

For simplicity, only two of the statements of the `p_activate` method are shown in the example below.

```
p_activate
p_activateCycleIndex :=
      ((p_activateCycleIndex + 1) \\ p_activateTally).
"Determine which statement should be executed."

(p_activateCycleIndex = 0)
ifTrue: [timer p_runTimer: timerEventQ]          "statement 0"

ifFalse: [(p_activateCycleIndex = 1)
```

```
        ifTrue: [self p_categoriseAndAllocate]      "statement 1"

        ifFalse: [...
                 ]
        ]
```

The above example serves to illustrate another issue that needs to be addressed during the mapping procedure, viz. the handling of parallel messages to the pseudo-variable `self`. The statement containing a message to `self` may be mapped as shown in the example above, or it may be expanded, in which case the resulting statements are included in the mapping. The second option is given next.

```
p_activate
p_activateCycleIndex :=
      ((p_activateCycleIndex + 1) \\ p_activateTally).
"Determine which statement should be executed."

(p_activateCycleIndex = 0)
ifTrue: [timer p_runTimer: timerEventQ]          "statement 0"

ifFalse: [(p_activateCycleIndex = 1)
      ifTrue: [scAllocator p_categorise: inputQ
                  using: scContainer]            "statement 1"

      ifFalse: [ (p_activateCycleIndex = 2)
            ifTrue: [scAllocator p_allocate:
                        scContainer from: inputQ]
                                                 "statement 2"
            ifFalse: [...
                     ]
               ]
         ]
```

If the second option is used, the `p_activateTally` variable has to reflect the total number of statements after the expansion has taken place. The expansion is mandatory if the ALBEDO meta-object infrastructure is used to perform the parallel statement selection. This issue is explained in detail in Section 8.6.2.

A brief description of the functionality of each statement that is executed in the above methods is given in the corresponding SLOOP methods in Appendix B, Section B.2.

At this point it may seem that the SLOOP method introduces added complexity during the mapping phase, because additional variables and statements are required in the mapped program. However, in Section 8.6 it will be shown how the concept of **computational reflection** can be used to ensure that variables and statements that do not form part of the SLOOP class can be implemented in a metaclass.

8.3.2 Synchronous shared-memory architectures

As described in Section 8.2.2, each *component-part* of a parallel statement can be allocated to a separate processor. Execution of a SLOOP statement is performed as described in Chapter 4, Section 4.3.6.3. To recapitulate: All *if* clauses are evaluated first, followed by the evaluation of all message expressions representing arguments of other message expressions, as well as the evaluation of all message expressions that play the role of the receiver of a message. Only then are the assignments and/or outermost message expressions executed. If the *if* clause of any *conditional-component-part-list* evaluates to false, then no further computation is performed for the corresponding *component-parts*.

Execution is restricted to **one statement at a time**. If, for a specific statement, no *component-part* is assigned to a particular processor, then that processor is idle for the duration of that statement. Although other mappings to synchronous shared-memory architectures are possible (e.g. where processors are not left idle), the simplicity of this mapping makes it easier to reason about its correctness. Thus, even though the *component-part*s of a second statement might not reference any of the objects referenced by the first statement, the second statement is not executed while the first is still busy executing.

Each *component-part* on each processor has to execute infinitely often. Furthermore, it has to be ensured that the *component-part*s belonging to the same statement execute **simultaneously** on the relevant processors. As a result, all the processors comprising the system have to execute the statements of the SLOOP program in the **same order**. The *component-part*s allocated to a specific processor are therefore enclosed within an infinite loop in a specific order on that processor.

8.3.3 Asynchronous shared-memory architectures

This section describes **a special case of an asynchronous shared-memory architecture**, viz. one where **multiple processes run on a single processor**. The processes share a common address space. Whereas in the case of a **synchronous** shared-memory architecture the *component-part*s of a parallel statement are assigned to different processors, here **complete statements** are assigned to processes.

Note that a parallel statement that is assigned to a specific process may invoke other parallel or sequential methods. In that case the statements comprising those methods are also executed by the same process. A process may therefore contain parallel statements belonging to multiple objects. Typically, the parallel statements belonging to a specific object are all assigned to the same process. However, that is not always the case, as will be seen below when the mapping of the `p_activate` method of the `CC_SimulationActivation` class is discussed.

Only parallel statements are assigned to processes. The statements of a sequential method are executed by the process containing the parallel statement which invoked the sequential method.

The following is an example of how the SLOOP parallel statements of the call centre program can be assigned to multiple Smalltalk processes running on the same Smalltalk virtual machine. Any number of Smalltalk processes can be used. In this example 6 processes are created. The statement allocation is summarized as follows:

Process number	Statements
1	No parallel statements
2	Parallel statements that invoke the parallel methods of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes.
3	Parallel statements that invoke the parallel methods of the `TimerServices` class
4	Parallel statements that invoke the parallel methods of the `ServiceCategoryAllocator` class
5	Parallel statements that invoke the parallel methods of the `ServiceCategory` class
6	Parallel statements that invoke the parallel methods of the `Connection` class

Table 8-1. Statement allocation to Smalltalk processes.

Process 1 contains the mapping of the **sequential** statements of the *activation-section*. It instantiates the necessary classes via the `setup` method of the `CC_SimulationActivation` class. It also creates all the other processes.

The purpose of the **parallel** statements in the *activation-section* is to ensure that the necessary parallel methods of the various classes are invoked infinitely often. In the mapping to the **sequential architecture**, this is achieved by enclosing the statement that sends the `p_activate` message to the `CC_SimulationActivation` instance in an infinite loop.

In the mapping to the **asynchronous shared-memory architecture**, the parallel statements of the `p_activate` method need to be **spread** over the various processes. For example, the statement that invokes the parallel method of the `TimerServices` class appears in process 3 and the ones that invoke the parallel methods of the `ServiceCategoryAllocator` class are present in process 4. The `p_activate method`, which merely served as a convenient way to group all of these statements in the SLOOP program, is therefore not used in this mapping. The parallel statements contained within the `p_activate` method are executed directly from within the *activation-section*. Statements containing messages to `self` are also **expanded** before the allocation of objects to processors is made. The mapping of the parallel statements of the *activation-section* is therefore spread over the various processes. In each process these parallel statements are enclosed in an infinite loop.

The Smalltalk-80 statements of process 1 are as follows (for brevity only the statements for invoking the parallel methods of the `TimerServices` and `ServiceCategoryAllocator` classes are shown):

```
| aCC_SimulationActivation |

aCC_SimulationActivation :=
      CC_SimulationActivation setup.

[ [true] whileTrue:
      [...] ]fork.
      "Process 2: Invoking the parallel methods of the
      CommsProviderSimulator and ServiceProviderSimulator classes."

[ [true] whileTrue:
      [timer p_runTimer: timerEventQ] ]fork.
      "Process 3: Invoking the parallel methods of the TimerServices
      class."

[ [true] whileTrue:
      [scAllocator p_categorise: inputQ using: scContainer.
      scAllocator p_allocate: scContainer from: inputQ
      ] ]fork.
      "Process 4: Invoking the parallel methods of the
      ServiceCategoryAllocator class."

[ [true] whileTrue:
      [...] ]fork.
      "Process 5: Invoking the parallel methods of the ServiceCategory
      class."

[ [true] whileTrue:
      [...] ]fork.
      "Process 6: Invoking the parallel methods of the Connection
      class."
```

Discussion:

Since the address space is shared by all processes, the `CC_SimulationActivation` class can be used to create all the instances that need to be present after initialization (via its `setup` method). Thereafter the various processes are created. The statements that belong to a particular process are enclosed in a Smalltalk block[4] and the message `fork` is sent to the block. All processes in this example are created at the same priority level. As soon as the message `fork` has been sent to a block, the process associated with that block becomes part of the list of processes that are scheduled by `Processor`, the single instance of the Smalltalk library class `ProcessorScheduler`. The latter is responsible for scheduling processes in a Smalltalk-80 system [GoRo89].

Thus, in the example shown above, the `timer p_runTimer: timerEventQ` statement which invokes the `p_runTimer:` parallel method of the `TimerServices` class, is enclosed in a Smalltalk block. The process that was created when the message `fork` was sent to the block, will eventually be scheduled and at that time the `timer p_runTimer: timerEventQ` statement will be executed by that process.

It was stated in Chapter 4 that a parallel **method** always returns after the execution of **one** of its parallel statements. Its statements are executed infinitely often by virtue of the fact that the **method** is **invoked** infinitely often. In contrast, the parallel statements in the *activation-section* of the program are not enclosed in a method. They are executed infinitely often because that is what the semantics of a parallel statement in the *activation-section* imply. In the mapping to the Smalltalk environment, it is therefore desirable to restrict infinite loops to the parallel statements in the *activation-section*. The parallel methods are invoked infinitely often via the infinite loop(s) in the *activation-section*.

In the mapping to the sequential architecture there was only one infinite loop in the *activation-section*. For the mapping to the asynchonous shared-memory architecture, there is an infinite loop for each process.

As stated earlier in this section, a parallel statement may invoke other parallel or sequential methods. For example, the `timer p_runTimer: timerEventQ` statement invokes the `p_runTimer:` parallel method. This method contains three parallel statements[5], as can be seen below:

```
        currentTime := SmalltalkLibPkg:::Time now asSeconds
      [] lastTime := currentTime \+
        currentTick := (currentTick + 1) \\ (timeoutCollection size)
            if difference ≥ 1 and: [currentTimeoutElement isNil]
      [] timerEventQ addLast: currentTimeoutElement \+
        currentTimeoutElement updateEndTime \+
        currentTimeoutElement timerServicesCompleted: true \+
        (timeoutCollection at: readIndex) removeFirst
            if currentTimeoutElement notNil
```

The above method is called a leaf parallel method[6], because the statements it contains do not invoke any parallel methods. Since only one SLOOP parallel statement contained in a leaf parallel method should be executed at each invocation of the method, it is necessary to insert a

[4] A Smalltalk block (delimited by square brackets) is defined as "a description of a deferred sequence of actions" in [GoRo89]. If the message `fork` is sent to a block, then a new process is created containing the expressions enclosed by the block.
[5] Details of the p_runTimer: method of the TimerServices class are given in Appendix B, Section B.11.
[6] In Chapter 4, Section 4.3.5.4, a leaf parallel method is defined as a parallel method that contains parallel statements invoking sequential methods only.

`Processor yield`[7] statement after each statement **within** the leaf parallel method. The process therefore relinquishes control after the execution of each parallel statement, enabling the `Processor` to schedule another process running at the same priority level.

In order to guarantee that the mapping to the above asynchronous shared-memory architecture retains the semantics of the original SLOOP program, the following aspects therefore have to be checked:

First of all, **each parallel statement** of the SLOOP program has to be represented by a Smalltalk statement which is **executed infinitely often**. This is achieved by assigning the parallel statements of each object to a process, and enclosing the statements of each process in an infinite loop. The parallel statements of an object may be assigned to a process either explicitly or implicitly. In the above example the statement `timer p_runTimer: timerEventQ` of the `p_activate` method of the `CC_SimulationActivation` class was allocated **explicitly** to process 3. However, the statements of the `p_runTimer:` method of the `TimerServices` class were allocated **implicitly** to process 3. Thus, all parallel and sequential methods invoked by a parallel statement are implicitly allocated to the process containing the invoking parallel statement.

Secondly, it has to be ensured that each SLOOP parallel statement contained in a leaf parallel method executes **atomically**. Thus, such a statement has to complete its execution before another parallel statement may be executed. This is achieved by disallowing any Smalltalk methods that relinquish control in the **mapped** statements. Thus, the Smalltalk counterparts of the SLOOP statements may not contain any message expressions that would relinquish control to the `ProcessorScheduler` instance.

However, each mapped parallel statement in a **leaf** parallel method is **followed** by a `"Processor yield"` statement in order to ensure that no process will forever **prevent** any other processes of the same priority from running. That is also the reason why all the processes are created at the same priority. It must also be guaranteed that each statement within the infinite loop in each process will **terminate,** i.e. no infinite loops are allowed in any of the statements enclosed by the outermost infinite loop of the process.

Since only one parallel statement in a leaf parallel method can be executed at a time (there is only one processor and each parallel statement in a leaf parallel method executes atomically), it is not necessary to reserve any objects prior to the execution of a parallel statement.

The third issue that has to be checked is whether each parallel statement in each leaf parallel method will indeed get a turn to be executed. Since only one parallel statement is executed at each invocation of a parallel method, the mapping of such a method has to ensure that **each parallel statement** contained within that method **will eventually be executed**. One possiblility is to introduce auxilliary variables in order to keep track of which statement should be executed next. An example of the usage of such variables was given in Section 8.3.1.

8.3.4 Distributed systems

This section describes the issues that are at stake when a SLOOP program is mapped to a distributed system. In section 8.2.4 the **allocation of statements and objects** to processors in a distributed system was discussed. This section continues that discussion with the focus on **retaining the semantics** of the SLOOP statements when they are mapped to a distributed architecture.

---

[7] When the message `yield` is sent to the ProcessorScheduler instance, the latter is instructed to give other processes at the priority of the currently running process a chance to run [GoRo89].

The basic premise is that the **correctness properties** that hold for a SLOOP program, will also hold for its mapped executable program, provided the mapping is done in such as way that the mapped statements reflect the semantics of the SLOOP statements. In order to achieve this, several issues need to be addressed:

- The atomicity of the SLOOP parallel statements must be preserved,
- the evaluation order of the SLOOP statements must be preserved,
- the semantics of the SLOOP message expressions must be preserved and
- the computational model must be preserved.

In the discussion below, the emphasis is on the last bullet. This is because it provides an opportunity to demonstrate how possible solutions to some of the complexities of a mapping can be **reused**.

The issues that need to be considered in order to address the first three points are therefore only discussed briefly in this introduction. As far as the preservation of the atomicity of the SLOOP statements is concerned, the following aspects are relevant: First of all, a SLOOP parallel statement is never spread over the processes in the distributed system. It is always a complete statement that is assigned to a process. However, a parallel statement may send messages to objects that reside at other processes. If two parallel statements share objects, they may not execute simultaneously. The (arbitrary) ordering of the execution of such statements will be discussed further when the issues related to the computational model are discussed below.

The evaluation order that needs to be preserved for SLOOP statements was first given in Chapter 4, Section 4.3.6.3 and summarised in Section 8.3.2 of the present chapter. Since a complete statement is mapped to a process in a distributed architecture, there is nothing specific to a distributed architecture that needs to be noted in this regard.

The preservation of the semantics of the message expressions is affected more by the target programming language than by the target architecture. However, in the case of a distributed architecture one also needs to ensure that the infrastructure that is used to transfer messages between objects guarantees the delivery of those messages and that it also guarantees that they will be delivered in the correct order.

As stated above, the remainder of this section focuses on issues related to the SLOOP computational model. In order to ensure that each mapped parallel statement will execute infinitely often, they have to be enclosed in an infinite loop. Furthermore, it has to be guaranteed that a mapped statement will not forever prevent another statement from executing, i.e. each mapped statement will eventually terminate. No statement should therefore contain an infinite loop. The only infinite loop that is allowed, is the outermost loop in **each process** that ensures that each statement is executed infinitely often. If there are multiple processes per processor, then each mapped parallel statement should also be followed by a statement which will yield control to the process scheduler. Absence of deadlock[8] should also be guaranteed.

The remainder of this section covers various aspects related to the prevention of deadlock in a mapping to a distributed architecture. The first subsection focuses on the rules that need to be followed regarding the reservation of objects in order to ensure that deadlock will not occur. The second subsection deals with the identification of the objects that need to be reserved and the third subsection describes why the CORBA Concurrency Control Service [OHE97] is not used to handle the object reservation aspect of the mapping to distributed architectures.

[8] Conditions for deadlock and ways of preventing deadlock as presented in the literature were discussed in Chapter 4, Section 4.3.6.5.

8.3.4.1 Guaranteeing absence of deadlock in a mapping to a distributed architecture

As mentioned in the introduction, this description of the mapping of SLOOP programs to distributed architectures focuses on the role of the SLOOP computational model. In this section it is demonstrated how the SLOOP approach enables the system designer to work at a **high level of abstraction**. Recall that during the design phase, the system is designed in terms of a number of **atomic** parallel statements, each executing infinitely often. The designer may **rely** on this atomicity at the design level, thereby **simplifying** the correctness reasoning at that level. There is no need to be concerned with complicated mechanisms to ensure exclusive access to objects and to guarantee that critical sections are handled correctly. The designer merely includes all the actions that need to take place atomically in a single parallel statement.

When the SLOOP program is mapped to its target environment, this **atomicity** has to be **retained**. At the same time, the mapping also has to guarantee that each mapped parallel statement will execute infinitely often. Thus, no statement should ever prevent any other one from executing. Once an **infrastructure** has been developed which satisfies these requirements, it can simply be **reused** by each subsequent mapping. Note that this infrastructure is different from the middleware infrastructure discussed earlier. The latter provides services such as locating the various objects in the system and transforming the messages into a format that can be transmitted over a communication medium. The SLOOP infrastructure discussed here **uses** the services of a middleware infrastructure.

In the discussion that follows, the required SLOOP infrastructure comprises a system which controls the sequence in which parallel statements at various processors may execute. The infrastructure determines which objects are referenced as target objects by each statement, it requests exclusive use of those objects on behalf of each statement and implements an algorithm which determines in what order the exclusive access may be granted. A distributed resource allocation algorithm is used in the example below. The remainder of this section is devoted to a description of the functionality of such an infrastructure.

*Note: In the description that follows, it is assumed that the distributed system consists of multiple processors, with a **single process** running on each processor. At the end of this section the impact of having **multiple processes** per processor will be discussed.*

Issues to be considered:

In the architectures described earlier, it was only necessary to be concerned with the allocation of **statements** to processes/processors. The mapping of a SLOOP program to a distributed system **also** involves the allocation of **objects** to processors, since there is no shared memory amongst the processors. This means that the data of the object is stored in the private memory of the processor and its methods are executed by that processor. Since there are multiple processors that execute concurrently and an object may receive a message from one remote object while it is busy processing a message from another remote object, the issue of ensuring the **integrity** of objects has to be addressed.

The integrity of an object can be ensured if there is **no interference**. This can be achieved by requiring that **an object** only executes methods related to a single parallel statement at a time. The **sequential**[9] methods invoked by a parallel statement are invoked **synchronously**[Vino97]. Nested upcalls[10] are allowed, but only if the upcalls are related to the execution of the current

[9] Since parallel methods contain parallel statements and each parallel statement must execute infinitely often, the correctness properties that are defined for the parallel methods take into account that multiple parallel methods of that object will execute concurrently.
[10] Synchronous invocation and nested upcalls were defined and discussed in Chapter 3, Section 3.2.2.1.

(local or remote) parallel statement. If two parallel statements **share objects**, then only the methods related to **one** of these parallel statements may be executed at a time. This restriction ensures that the correctness arguments used during the design phase are preserved during the implementation phase. When a sequential method is executed, **the state of the object is defined by the total correctness property of the method being executed**. If the above restriction is adhered to, no other method of the object can interfere with the state of the object during that execution. In Chapter 4 this requirement was illustrated by two scenarions. In Figure 4-8(a) it was shown that the nested upcalls belonging to the same parallel statement would always result in the same execution sequence of the sequential statements, whereas the nested upcalls belonging to two different parallel statements sharing objects and executing simultaneously could result in arbitrary execution sequences.

However, by restricting execution at a specific processor to the methods belonging to a single parallel statement at a time, a new problem is introduced: it is possible for deadlocks to occur (a scenario for deadlock in such a situation is described in Chapter 3, Section 3.2.2.1), unless preventative measures are taken. In [Tane92] the reservation of resources is given as one possible mechanism to prevent deadlock. Thus, by ensuring that all the relevant objects are reserved for the exclusive use of a particular parallel statement prior to the commencement of the execution of that statement, deadlock can be prevented. The relevant objects are the object to which the parallel statement belongs, as well as all the target objects[11] referenced either explicitly (in the statement itself) or implicitly (when a message is sent to an object from within one of the methods that are invoked as part of the chain of methods that are executed by the statement).

Apart from guaranteeing the **integrity of a SLOOP object**, this solution also ensures that there is no **interference** when parallel statements are executed. A SLOOP parallel statement always contains a modifying part, optionally governed by a conditional part. The SLOOP model relies on the fact that the state of an object does not change between the time that the conditional part is evaluated until the modifying part is executed. While a parallel statement is being executed, no other parallel statement should interfere with the states of the objects referenced by the former.

Recall that the discussion in this section focuses on the case where **each processor** in the distributed system has only **one process** running on it. In the deadlock prevention strategy described below, it is assumed that each parallel statement **within a process** must execute to completion before the next parallel statement within that process can commence execution. Thus, if parallel statement *sA1* at processor A requires object *oA1* at processor A and object *oB1* at processor B, while parallel statement *sA2* within the **same** process requires object *oA2* at processor A and *oB2* at processor B, then the execution of statement *sA2* does not commence before the completion of statement *sA1* if the latter is currently being executed. This is because the atomic unit of execution **within a process** is a parallel statement. Although the parallel statements may be executed in an arbitrary order, each statement is executed to completion before the execution of the next statement is commenced.

This requirement is extended further to include the parallel statement which handles the messages received as a result of the execution of parallel statements located at remote processors. Thus, while the process at processor A is busy executing statement *sA1* and it is waiting for a response to its message sent to object *oB1* (which resides at processor B), the process at processor A is blocked except for nested upcalls belonging to the blocking statement. Thus, the process will only respond to messages received via the execution of statement *sA1*.

If **multiple processes** are implemented **at each processor**, the **efficiency** of the implementation can be improved considerably. In that case the parallel statement which receives messages from processes at other processors could be assigned to a separate process. It will then be allowed to

[11] A target object is an object to which a message is being sent. The client object is the object sending the message.

execute concurrently with the parallel statements in the other processes at that processor, provided the statements do not share objects. However, such improvements are not discussed here, since the purpose of this section is merely to describe one possible way of mapping a SLOOP program to a distributed architecture in order to be able to highlight the **reusable** aspects of the mapping.

The deadlock prevention strategies described below are therefore aimed at an architecture comprising multiple processors, but only one process per processor. They are based on the simple conceptual model that each parallel statement within a process executes to completion before any other parallel statement within that process is executed. This includes the parallel statement which handles messages from remote parallel statements.

This approach has the following implication: Even if two parallel statements at two different processors do not send messages to the same objects, deadlock could still occur if they send messages to objects that share **processors**. The scenario shown in Figure 8-1[12] supports this claim: Statement *sA1* at processor A requires objects *oA1* and *oB1*, while statement *sB1* at processor B requires objects *oA2* and *oB2*. Thus, there are no shared objects. However, the statements send messages **synchronously** to objects that share processors (*oA1* and *oA2* share processor A while *oB1* and *oB2* share processor B).

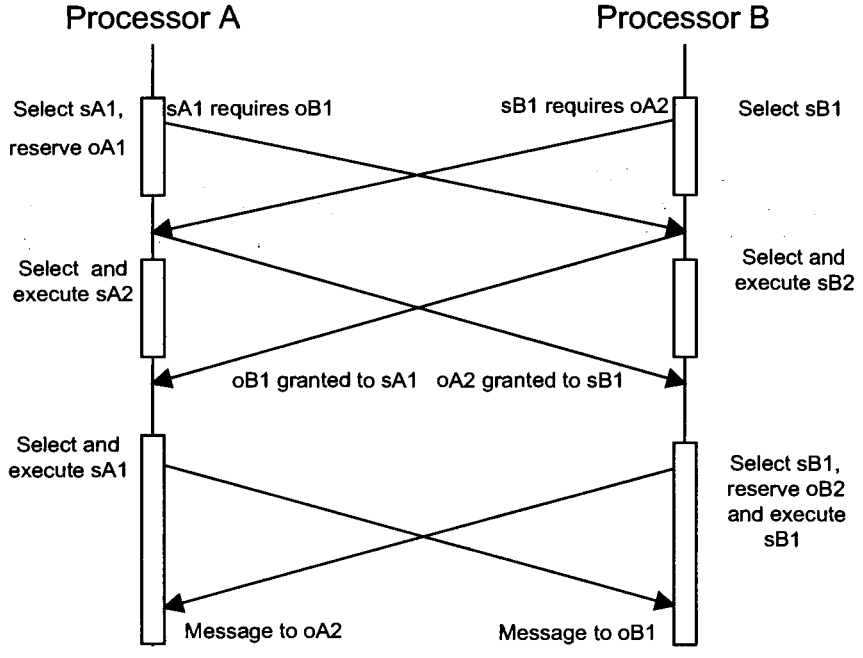

Figure 8-1. Scenario for deadlock when there are no shared objects, but the objects share processors in a single process per processor distributed architecture.

[12] The processors are identified via letters of the alphabet. The statements and objects at the various processors are identified by numbers prefixed by the letters designating the relevant processor, as well as an 's' to indicate a statement or an 'o' to indicate an object respectively. As is evident from Figure 8-1, reservation requests and confirmations are handled by each processor in between the execution of statements. Note that a statement is **selected** for execution on a round robin basis, but is only **executed** if all its reservation requests have been granted. At each processor one parallel statement is dedicated to the handling of messages from remote clients, i.e. it receives the messages from the clients and then passes them on to the relevant local objects. In Figure 8-1 this function is performed by statements *sA2* and *sB2*.

If the reservation requests are granted, deadlock occurs if the following happens: Statement *sA1* starts executing and sends a message to object *oB1* while at the same time statement *sB1* starts executing and sends a message to object *oA2*. Since the messages are being sent synchronously, the process at processor A blocks while waiting for the message to object *oB1* to complete, while the process at processor B blocks while waiting for the message to object *oA2* to complete.

The mapping of a SLOOP program to a distributed architecture where only **one process runs on each processor** therefore has to ensure the following:

- Each process only executes **one parallel statement at a time** and it executes it to **completion** before proceeding to the next one. Thus, parallel statements that are located at the same processor have to be executed in some (arbitrary) order.
- When two parallel statements are located at different processors and they **share objects**[13], the parallel statements have to be executed in **some (arbitrary) order**.
- Parallel statements that are located at different processors and that **do not share objects** (either explicitly or implicitly) have to be executed in **some arbitrary order** if the **target objects**[14] referenced by the different statements **share processors**.

Parallel statements may therefore execute simultaneously if the **statements are located at different processors,** they **do not share objects** (either explicitly or implicitly) and the **target objects** referenced by the respective statements **do not share processors**.

Addressing the issues:

One way of achieving all of the above is to acquire **all the resources** pertaining to a particular atomic execution **prior** to the commencement of that execution. This will automatically impose a sequential ordering on the statements that share resources. Based on the above discussion, it would appear as if these resources are the processors rather than the objects, because any statements that refer to target objects sharing the same processor may not execute simultaneously, even if these statements do not share any objects.

However, by viewing the processors as the resources that have to be reserved, the scope for concurrency becomes very limited. For that reason the target objects that are referenced by a statement are viewed as the resources, but special rules apply regarding the reservation of these resources. Details regarding these rules are given below.

The general strategy for object reservation:

Before providing detail regarding the resource reservation algorithm used here, the general strategy is described first. One of the aims of this algorithm is to ensure that no statement will be prevented forever from executing as a result of resource allocations to other objects, i.e. **each statement will eventually be granted its required resources and be allowed to execute**. Furthermore, while resources are being allocated to a statement, other statements will only be prevented from execution if their execution will result in the violation of the correctness properties of the system. Thus, **concurrent execution** must be **maximised** as far as possible.

---

[13] Two parallel statements share objects if they send messages to the same objects, or if the two parallel statements belong to the same object or if the one parallel statement sends a message to the object to which the other parallel statement belongs. Objects may be shared either explicitly via references in the parallel statement itself, or implicitly via references within the methods invoked by the parallel statement.

[14] If an object is not the receiver of the parallel statement under consideration, or it does not act as a target object during the execution of the parallel statement, but it is merely referenced (for example its value is assigned to a variable), then that object does not have to be considered when identifying the objects that could affect the possibility of deadlock. This is because none of the statements of such an object are executed during the execution of the parallel statement under consideration.

Requesting resources:

When a parallel statement is selected for execution, the location of each target object referenced by that statement is determined. A single reservation request is composed for **each** target processor. All the objects required from the specified processor for that particular parallel statement are listed in the reservation request destined for that particular processor.

The specified objects at the specified processor are allocated to the parallel statement in a single atomic action. There is therefore no need to reserve the **objects** located **at a particular processor** in a specific order. However, reservation requests are sent to **processors** in a prescribed order, since it cannot be guaranteed that all the resources pertaining to a specific statement will be granted simultaneously at all the processors involved. Ordering of requests removes the circular wait condition of deadlock [Tane92] and therefore prevents the object reservation algorithm itself from running into a deadlock situation.

For example, processor A has acquired objects from processor B and is now requesting objects from processor D, while processor C has acquired objects from processor D and is now requesting objects from processor B. Both A and C will wait forever for the outstanding resources as the resource allocation graph[15] [Tane92] in Figure 8-2(a) illustrates. If requests are always issued in some (arbitrary) order, deadlock is prevented, as shown in Figure 8-2(b). The request sent to the first processor in the sequence therefore has to be granted before the request to the second processor may be issued. In the example in Figure 8-2(b) processor C can only send its request to D once its request to processor B has been granted.

Note that if a statement sends messages to **local objects only and none of those objects have been allocated to or requested by other parallel statements, no reservation request is issued**. The statement may execute immediately, since it is guaranteed to complete and it cannot interfere with the execution of any other statement.

If a statement sends messages to **local objects only, but one of the objects has already been allocated to a remote parallel statement, a reservation request has to be issued.** This is to prevent interference. For example, if parallel statement *sB1* at processor B has been granted exclusive access to object *oA1* and it starts executing, it might check the state of object *oA1* in the conditional part of statement *sB1*. However, if statement *sA1* (which sends messages to object *oA1* only) is allowed to execute before statement *sB1* can execute its modifying part, it could change the state of *oA1*. This implies that the atomicity requirement of statement *sB1* would be violated. It is for this reason that statement *sA1* has to issue a reservation request if it is found that it has to send messages to objects that are currently allocated to or have been requested by a remote parallel statement.

If a statement sends messages to **both local and remote objects**, reservation requests are issued for the local objects as well as for the remote objects in the prescribed order.

[15] In a resource allocation graph processes are represented by circles and resources are shown as squares. If an arc is directed from a resource towards a process, then the resource has been granted to that process. If an arc is directed from a process towards a resource, then the process is waiting for that resource. A cycle in the graph indicates deadlock.

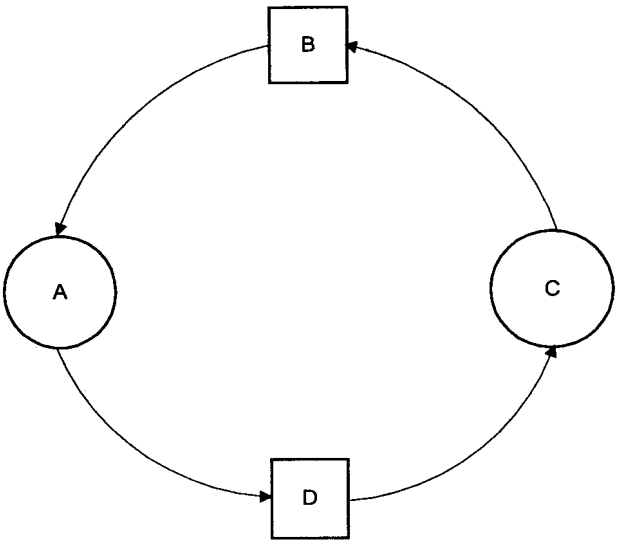

Figure 8-2(a). Resource allocation graph showing deadlock.

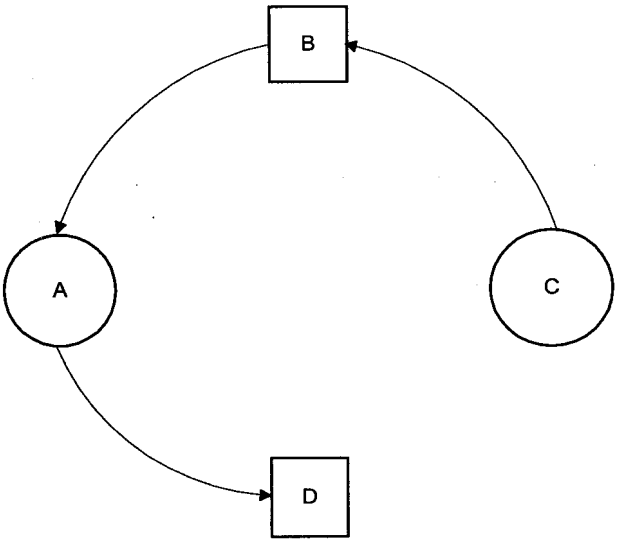

Figure 8-2(b). Deadlock is prevented if the resource allocation requests are made in an (arbitrary) order.

Granting resources:

When a request for the reservation of object(s) at a particular processor has been granted to a parallel statement (regardless of whether that statement is local or remote), all other reservation requests for objects at that processor are queued by the specified processor if the reservation requests are received from **remote** processors. The behaviour when the reservation request pertains to a local parallel statement will be described shortly. Once one or more of the objects at a processor have been allocated to a parallel statement, no other objects located at that processor are allocated to **remote** parallel statements until the resources have been released.

The rationale for this restriction is as follows: Once a resource has been granted to a remote processor, the parallel statement at the remote processor could start executing whenever that statement has acquired all its resources. If a local parallel statement has also acquired all its resources in the meantime, both statements could start executing simultaneously. This is because

the logic determining which statement can be executed next at a particular processor does not take the status at any remote processor into account. The only criterium that is used to decide whether a statement should start executing is whether all the required resources have been granted. As a result deadlock could occur if the parallel statements refer to target objects that share processors, since the method invocations are synchronous. This is demonstrated by the scenario depicted in Figure 8-3:

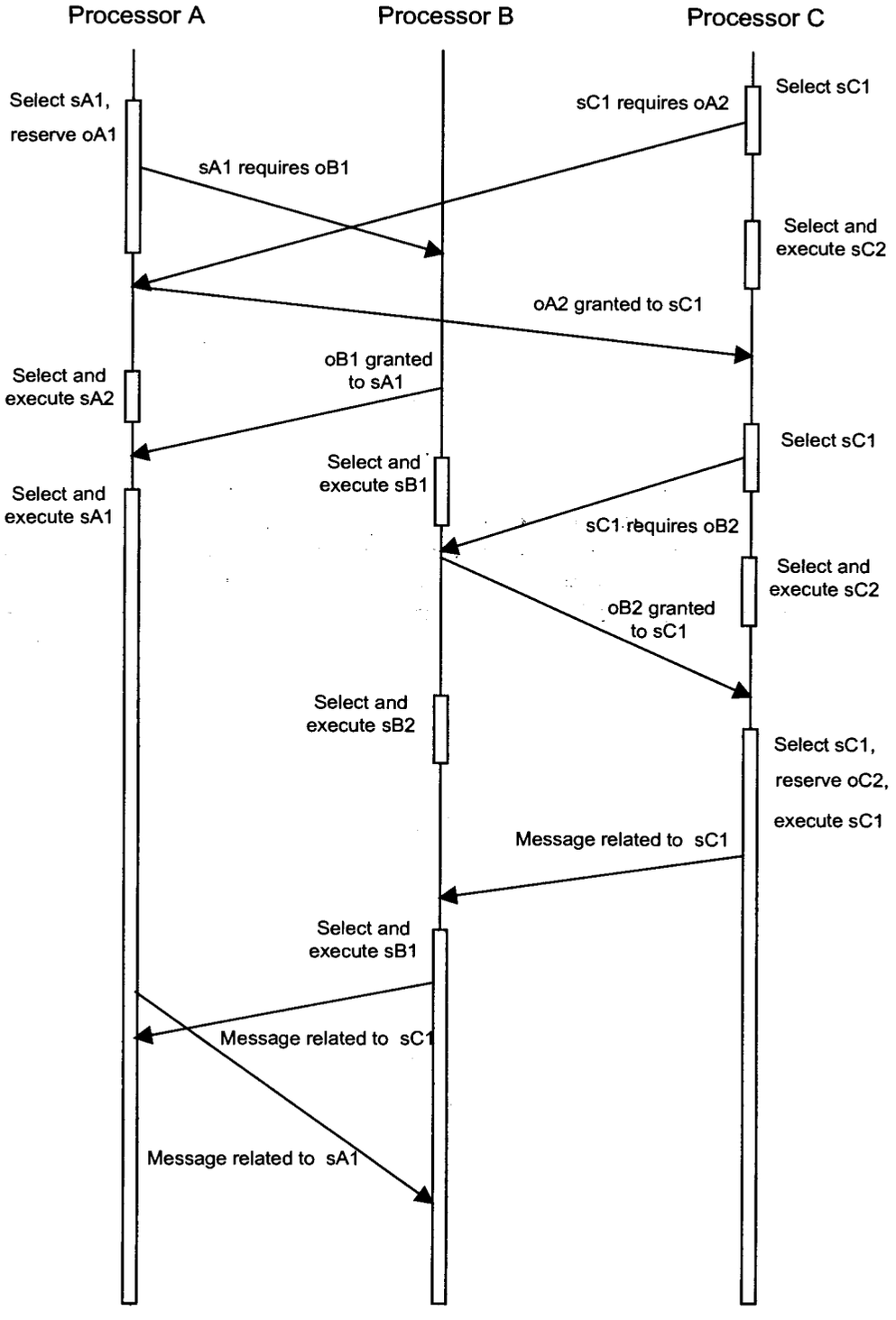

Figure 8-3. Deadlock as a result of the granting of multiple reservation requests.

Statement *sA1* at processor A requires objects *oA1* and *oB1* at processors A and B respectively. Statement *sC1* at processor C requires objects *oA2*, *oB2* and *oC2* at processors A, B and C respectively. All requests are granted, since there are no conflicts. Statement *sA1* starts executing. It sends a message to *oB1* and waits for a response. In the meantime statement *sC1* has also started executing. It has sent a message to *oB2*. Processor B is currently handling that message, which requires a message to be sent to *oA2*. Deadlock ensues, since processor A is still waiting for a response to the message sent to *oB1*.

If the restriction is adhered to, then the request for object *oA2* to be allocated to statement *sC1* is not granted by processor A. This is because a local object (*oA1*) has been reserved for a local statement that sends messages to both local and remote objects. As a result, when the request for object *oA2* is received, it is queued until statement *sA1* has released object *oA1*. Deadlock is therefore prevented.

In **more generic terms**, the above algorithm merely ensures that deadlock is prevented at the **processor level,** i.e. if two statements send messages to objects that share processors, then those statements are forced to execute in an (arbitrary) order. Thus, the circular wait condition for deadlock is eliminated [Tane92].

A **two-tiered approach** is therefore followed to prevent deadlock. **Requests** are made for exclusive access to the relevant **objects** for a particular statement. This allows statements to be executed in parallel if such concurrency will not compromise the correctness of the system, as will be seen below when the requests pertaining to local parallel statements are discussed.

On the other hand, when requests are **granted**, it is done at a per **processor** level if the request is received from a **remote** processor. If the reservation request pertains to a local parallel statement, more concurrency is possible, as will be seen shortly.

Note that even when a processor **grants a reservation request** to a remote parallel statement, it does **not** mean that the specified processor is **blocked**. The processor continues to execute its infinite loop. It will therefore continue to execute any local parallel statements that have been granted access to all the required objects. When a parallel statement that has reserved objects has **completed its execution**, all the objects that are reserved for that statement are **released** for allocation to other statements.

The treatment of **local reservation requests** is discussed next. Note that a local reservation request is only issued if a local parallel statement sends messages to both local and remote objects or if the local parallel statement sends messages to object(s) that have been allocated to or requested by another parallel statement. The reason for treating local reservation requests differently from remote reservation requests is as follows: If two local parallel statements do not share any resources, then the **efficiency** of the algorithm can be improved by allowing the resources for these statements to be **reserved** in parallel. Since these statements share the local processor, they will **not be executed** in parallel. The correctness of the system is therefore not compromised by the concurrent **reservation** of the resources required by these statements.

When a reservation request for a resource is issued for a local parallel statement, the request is queued if **any** local object has been allocated to a **remote** parallel statement or if any requests from remote parallel statement(s) have been queued. This is done to prevent a deadlock situation. If the scenario shown in Figure 8-3 is modified such that the request for object *oA2* reaches processor A before object *oA1* is reserved, then the request for object *oA2* will have been granted when the reservation request for object *oA1* is made. Thus a request for a local object is made when a local object has already been allocated to a remote parallel statement. It is clear that if the last request is granted, then exactly the same deadlock situation as depicted in Figure 8-3 is possible.

If no remote **statements** are involved, the request is granted if **all** the local objects listed in the request can be allocated, otherwise the request is queued. For example, if object *oA2* in Figure 8-4(a) has been allocated to statement *sA1* (i.e. to a local parallel statement sending messages to both local and remote objects) and statement *sA2* requires objects *oA2* and *oA3* (in addition to remote objects), the reservation request pertaining to statement *sA2* is queued, as shown in Figure 8-4(a). Object *oA3* is not allocated, even though it is free, since all the objects required by a specific parallel statement are allocated in a single atomic action. This ensures that whenever a statement releases its resources, then all the objects will be available to the next request in the queue.

If a reservation request is now issued for statement *sA3* indicating that object *oA3* is required (in addition to one or more remote objects), the request is also queued. Although object *oA3* is free, it has already been requested by statement *sA2* and should therefore be allocated to statement *sA2* first.

However, if the only local object required by statement *sA2* had been object *oA2*, then the reservation request for statement *sA3* would have been granted, as can be seen in Figure 8-4(b). The concurrent acquisition of the resources required by statements *sA1* and *sA3* is thereby facilitated.

Thus, the purpose of allowing multiple **local** reservation requests to be granted simultaneously, is to allow multiple local parallel statements that do not send messages to the same objects to **acquire their resources in parallel**. These statements cannot interfere with each other, since they are never processed in parallel (owing to the fact that they share the same processor). Note that it is only necessary to check whether the statements share **local** objects. If they share remote objects, the algorithm executed at the remote processors will ensure that the remote objects will not be allocated to both statements simultaneously.

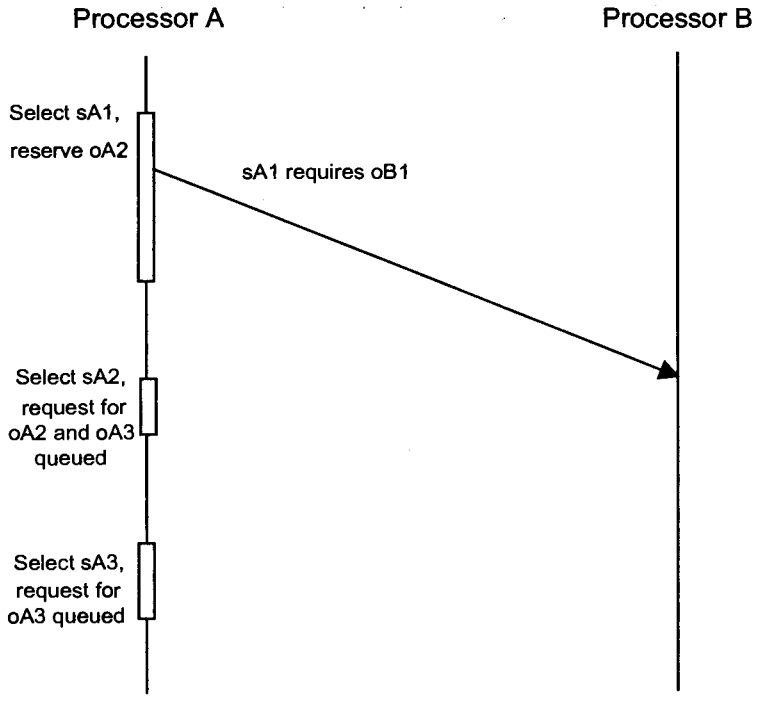

Figure 8-4(a). Handling of local reservation requests (shared local objects).

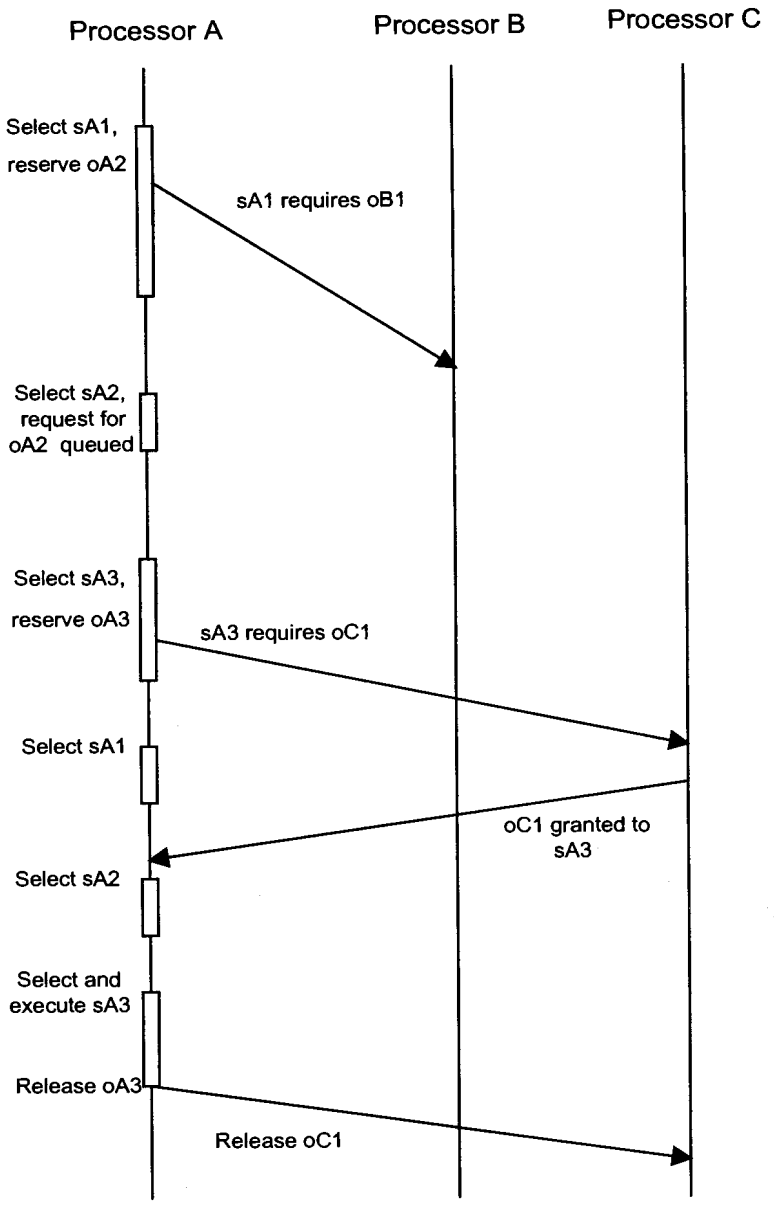

Figure 8-4(b). Handling of local reservation requests (no shared local objects).

It would be possible to achieve even more parallelism by allowing local parallel statements to obtain their remote resources concurrently regardless of whether they share local objects or not. This is because they will not execute simultaneously, since they share the local processor. However, that would complicate the resource allocation algorithm, since it would have to make provision for local objects being allocated to multiple local parallel statements at the same time. This option is not discussed further here, but this and other ways of improving the efficiency of the algorithm is a topic for further research.

The resource allocation algorithm:

At each processor in the distributed system, the following actions must be taken:

- Resource reservation requests received from remote processors must be handled.
- Each parallel statement in the infinite loop running on a processor must be selected infinitely often for execution. When a statement is selected for execution and all the relevant objects have already been allocated to it, it is executed. If there are still some objects outstanding, the necessary action is taken as described below. The statement is not executed.

- Indications that local objects have been released by remote parallel statements must be processed.
- Indications that remote objects have been allocated to local parallel statements must be processed.

A pseudo-code version of the basic functionality of the resource allocation algorithm is now presented. For easy reference, the different sections of the algorithm are referred to as rules and are numbered.

```
[
   (reservation request received from remote processor)                "Rule 1"
   ifTrue:
   [
      (any local object(s) already allocated to or requested by a
       local or remote parallel statement)
         ifTrue: [append the request to the local Request Queue
                  if it is not already present in the local Request Queue]
         ifFalse: [grant the request to the remote parallel statement]
   ]
   ifFalse:
   [
      (local objects released)                                         "Rule 2"
      ifTrue:
      [
         (all the local objects required by the first entry in the
          local Request Queue available)
         ifTrue:
         [
            grant the request to the entry in the local Request Queue.
            (the request is for a local parallel statement)
            ifTrue:
            [
               (all objects now reserved)
               ifTrue: [execute the statement and then release all
                        objects held by the statement]
               ifFalse: [issue the next request to a remote processor]
            ]
            ifFalse: [no further action taken]
         ]
         ifFalse: [no action taken]
      ]
      ifFalse:
      [
           (request granted at a remote processor)                     "Rule 3"
           ifTrue:
           [
              (access granted to all remote objects that should be reserved
               prior to the local objects)
              ifTrue:
              [
                 (any of the required local object(s) already allocated to
                  or requested by another local or remote parallel
                  statement)
                 ifTrue: [create a request and queue it in the local
                          Request Queue if it is not already present in the
                          local Request Queue]
                          "to prevent deadlock and interference"
                 ifFalse:
                 [
                    allocate the objects to the local parallel statement.
                    (all objects now reserved)
                    ifTrue: [execute the statement and then release all
                             objects held by the statement]
                    ifFalse: [issue the next request to a remote processor]
                 ]
              ]
```

```
                ifFalse: [issue the next request to a remote processor]
            ]
            ifFalse:
            [
                "next local parallel statement selected"
                (next local parallel statement requires local objects only)
                                                                    "Rule 4"
                ifTrue:
                [
                    (any of the required local object(s) already allocated to or
                     requested by a remote parallel statement)
                    ifTrue: [create a request and queue it in the local
                            Request Queue if it is not already present in the
                            local Request Queue]
                            "to prevent deadlock and interference"
                    ifFalse: [allocate the objects to the local parallel
                             statement and execute the statement]
                ]
                ifFalse:
                "next local parallel statement requires local and remote objects"
                                                                    "Rule 5"
                [
                    (access granted to all remote objects that should be reserved
                     prior to the local objects)
                    ifTrue:
                    [
                        (any of the local objects already allocated to or requested
                         by a remote parallel statement or
                         any of the required local object(s) already allocated to
                         or requested by another local parallel statement)
                        ifTrue: [create a request and queue it in the local
                                 Request Queue if it is not already present in the
                                 local Request Queue]
                                 "to prevent deadlock and interference"
                        ifFalse:
                        [
                            allocate the objects to the local parallel statement.
                            (all objects now reserved)
                            ifTrue: [execute the statement and then release all
                                 objects held by the statement]
                            ifFalse: [issue the next request to a remote processor]
                        ]
                    ]
                    ifFalse:
                    [
                        issue the next request to a remote processor
                    ]
                ]
            ]
        ]
    ]
]
```

As is evident from the above, the algorithm implements five rules:

Rule 1:
When **a reservation request** is received from a **remote** processor (the requesting processor), the request is **queued**[16] in the local Request Queue instead of granted if local object(s) are already allocated to or requested by a parallel statement. The latter may be a local parallel statement or one at another processor. It may even be another parallel statement at the requesting processor.

[16] A request is only queued if it is not already present in the local Request Queue. This applies to all the rules listed here.

Rule 2:
When **local** objects are **released** by a local or remote parallel statement, the next entry in the local Request Queue is inspected.

- If this entry is a request for local object(s) to be allocated to a parallel statement at a **remote** processor, the request is granted if all local objects are available. Otherwise the request remains in the queue until the remaining objects have also been released. This takes care of the requests queued as a result of Rule 1.
- **Else** if the entry is a request for local object(s) to be allocated to a **local** parallel statement, the request is granted if the requested objects are now free. This ensures that requests queued as a result of Rule 4 or 5 are processed. If all the required objects are now reserved, the statement is executed and then the objects are released. If all the required objects have not been reserved yet, the request to the next processor in the sequence is issued.

Rule 3:
When an indication is received that **remote objects have been reserved** for a local parallel statement, the algorithm checks whether there are any more objects that need to be reserved.

- If all objects have not been reserved yet and the next request in the sequence is one requesting remote objects, a request is sent to the relevant remote processor.
- **Else** if local objects should be reserved next, a request is **created and queued** in the local Request Queue if **any** of the required local objects are currently allocated to or requested by a **remote** parallel statement. It is also queued if the **specified** local object(s) are currently allocated to or have been requested by a **local** parallel statement.
- **Else** if all the requests pertaining to this statement have been granted, the statement is executed and then the objects are released.

Rule 4:
When the **next local parallel statement has been selected for execution** and all the **required objects** are **local**, a request is **created and queued** in the local Request Queue if **any** of the required local objects are currently allocated to or requested by a **remote** parallel statement. If all the required local objects are available, the statement is executed.

Rule 5:
When the **next local parallel statement has been selected for execution** and **both local and remote objects** are required, the following actions are taken:

- A request is sent to a remote processor if all objects have not been reserved yet and the next request in the sequence is one requesting remote objects.
- If local objects should be reserved next, a request is **created and queued** in the local Request Queue if **any** of the required local objects are currently allocated to or requested by a **remote** parallel statement. It is also queued if the **specified** local object(s) are currently allocated to or have been requested by a **local** parallel statement.
- **Else** if all the requests pertaining to this statement have been granted, the statement is executed and then the objects are released.

Thus, the **underlying goal** of this algorithm is to ensure that each statement in the infinite loop on each processor will always **terminate**[17], thereby ensuring that each parallel statement can be executed infinitely often. If all the local and remote resources required by a parallel statement have been granted to it according to the procedures specified above before it starts executing, its termination is guaranteed. **Separate** reservation requests are made for **each parallel statement**, since the latter is the atomic unit of execution.

[17] It is assumed that all the sequential methods invoked by the parallel statements are terminating methods.

Additional comments on the resource allocation algorithm:

- The entries of the local Request Queue are always processed on a strictly First In First Out basis.
- The processors are ordered in some (arbitrary) order and reservation requests for each parallel statement are issued strictly in this order.
- All requests for resources are always issued at the processor where the parallel statement is located. When a message is received from a remote object and the target object on the local processor needs to send a message to a remote object on yet another processor, it is therefore guaranteed that the execution will terminate, since all the resources required by the parallel statement on the originating processor have to be allocated to that statement before it starts executing.
- All resource requests are issued asynchronously, i.e. control is returned to the requesting processor without waiting for a response. Before any parallel statement is selected, the queue containing requests and responses from remote processors is examined. If it is not empty, its entries are processed.

- When multiple processors are involved, the sequential statements appearing in the *activation-section* of the SLOOP program are allocated to the various processors as appropriate. All of these sequential statements have to complete execution before the parallel statements start executing. One mechanism to achieve the desired behaviour is to prohibit the granting of object(s) to a remote **parallel** statement if all the sequential statements in the *activation-section* allocated to the specified processor have not yet completed execution. The requests are queued at the target processor. Note that the objects may be granted to a **sequential** statement appearing in the *activation-section* on a remote processor.

- If the order in which the sequential statements in the *activation-section* execute is significant, then the necessary synchronization mechanisms have to be included for this purpose during the mapping procedure.

Scenarios to illustrate aspects of the resource allocation algorithm:

Several scenarios are now presented to elucidate the above principles. Figure 8-5 illustrates how reservation requests are queued when the requested objects are not available. It also shows that reservation requests for a specific parallel statement are made sequentially. Only one reservation request may be outstanding at a time for a particular parallel statement.

There are three parallel statements at processor A. Statement *sA1* is selected for execution. It requires objects *oA1*, *oB1* and *oC1*. Object *oA1* is allocated to statement *sA1*. A request for object *oB1* is issued. Without waiting for a response from processor B, statement *sA2* is selected. It requires objects *oA2*, *oB1* and *oD1*. Object *oA2* is allocated to statement *sA2*. Object *oD1* is available. However, no request for object *oD1* is made, since it has not yet acquired object *oB1*. A request for object *oB1* is issued for statement *sA2* and queued at processor B. The third parallel statement (*sA3*) services messages that are received from remote objects, therefore it does not need to reserve any objects.

By the time that statement *sA1* is selected again, object *oB1* has been granted to it, therefore a request for object *oC1* is issued. The latter cannot be allocated to statement *sA1* yet, so the request is queued at processor C. Statement *sA1* is not executed, since an object request is still outstanding. Statement *sA2* is now selected for execution. No action is taken, since it is still waiting for object *oB1* to be granted to it.

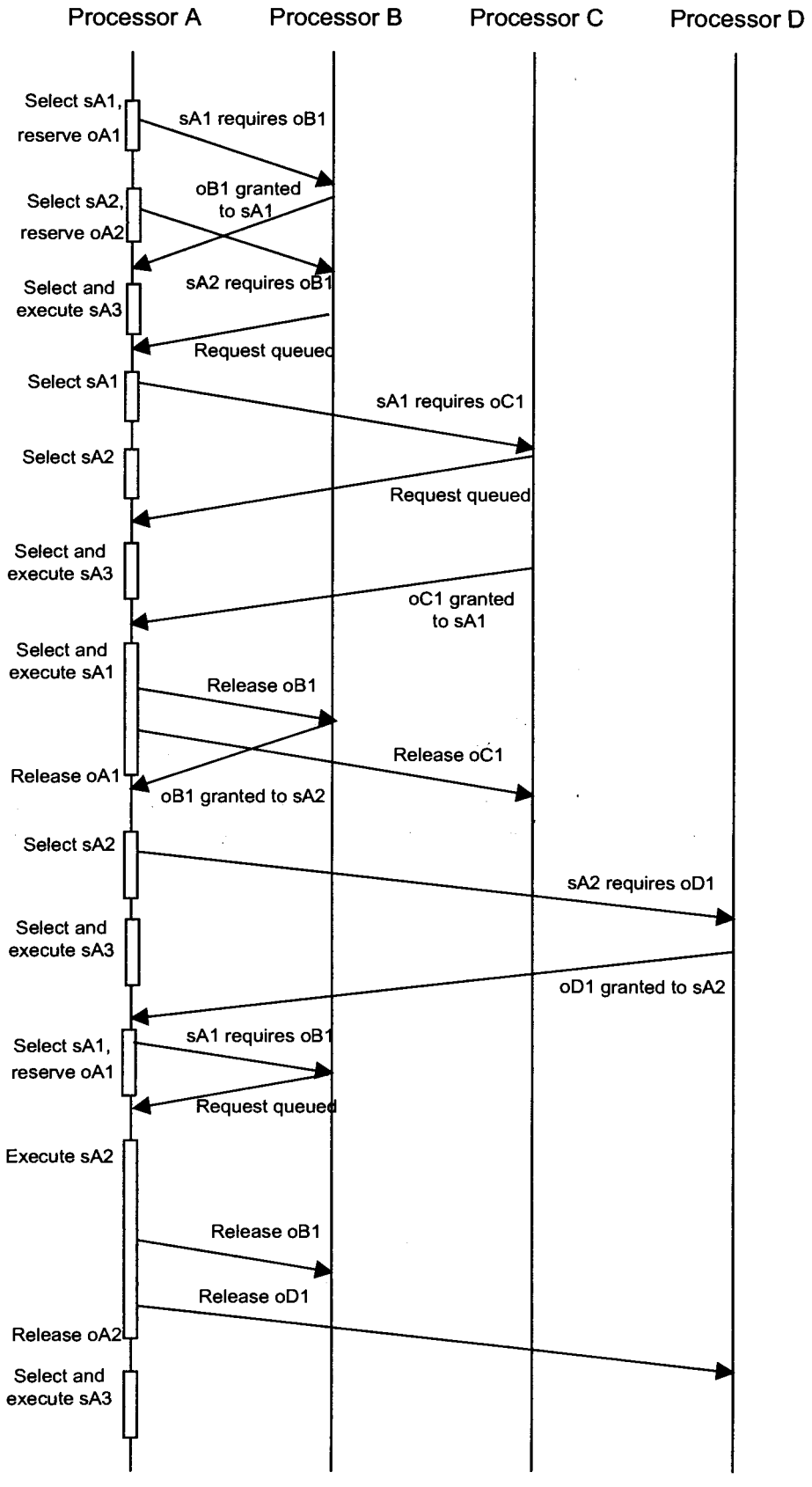

Figure 8-5. Scenario illustrating how resources may be requested by and granted to parallel statements at processor A.

Once the request for object *oC1* has been granted, statement *sA1* is executed. (The messages sent to the objects located at processors B and C are not shown in Figure 8-5.) When it has completed, it releases all its resources. The request for object *oB1* can now be granted to statement *sA2*. A request for object *oD1* is made on behalf of statement *sA2*. The latter only executes once it has acquired all its resources.

The next scenario illustrates how the resource allocation algorithm presented in this section allows for concurrency without introducing undue complexity. While local objects are allocated to parallel statements that require objects at multiple processors, the local processor may still execute local parallel statements that are unaffected by these object allocations. In particular, it may execute:

- the local parallel statement which services messages from remote objects,
- local parallel statements that do not send messages to remote objects and which do not require the local objects that are currently allocated to or requested by remote parallel statements, as well as
- local parallel statements to which all resources have been granted.

In the scenario depicted in Figure 8-6 processor B has two local parallel statements. The first one (statement *sB1*) only sends messages to objects *oB2* and *oB3* at processor B and the second one (statement *sB2*) services messages from remote objects. While object *oB1* is allocated to statement *sA1*, processor B is allowed to execute its local parallel statements that do not require any remote resources and that also do not require object *oB1*, as shown in Figure 8-6.

If the request for object *oC1* is granted and processor A starts executing statement *sA1*, a message related to statement *sA1* at processor A may arrive while processor B is executing statement *sB1*. Since the latter only sends messages to objects *oB2* and *oB3*, local objects that are currently not allocated to other statements, it is guaranteed to terminate. Processor B will therefore eventually execute its second parallel statement (*sB2*), which services messages from remote objects. That implies that the message from statement *sA1* will eventually be serviced.

Note that the statement which services messages from remote objects is always executed when it is selected, since no resources need to be reserved in order to execute this statement. By the time a message is received from a remote object, all the resources related to the statement to which this message belongs have already been reserved. For example, in the above scenario all resources required for the parallel statements on processor A are issued at that processor. Other processors never need to issue requests for resources on behalf of processor A. By the time statement *sA1* is allowed to execute, objects *oB1* and *oC1* are already allocated to it.

Thus, when processor B executes statement *sB2* (which results in the processing of the message related to *sA1*), it does not have to reserve any resources prior to the execution of the message. Even if the processing of this message results in a message being sent to an object at processor C, it is guaranteed that object *oC1* will be available to statement *sA1*, since it has been reserved for that statement by processor A.

If there is a fourth parallel statement (*sA4*) on processor A which only refers to objects *oA3* and *oA4* at processor A, that statement may also execute while statements *sA1* and *sA2* have not yet acquired all their resources. This is allowed, since a parallel statement which only refers to local objects that are not allocated to other remote statements at that moment, will never interfere with the state of an object that is allocated to another object. The statement will also always terminate, allowing the processor to execute the next statement in its infinite loop.

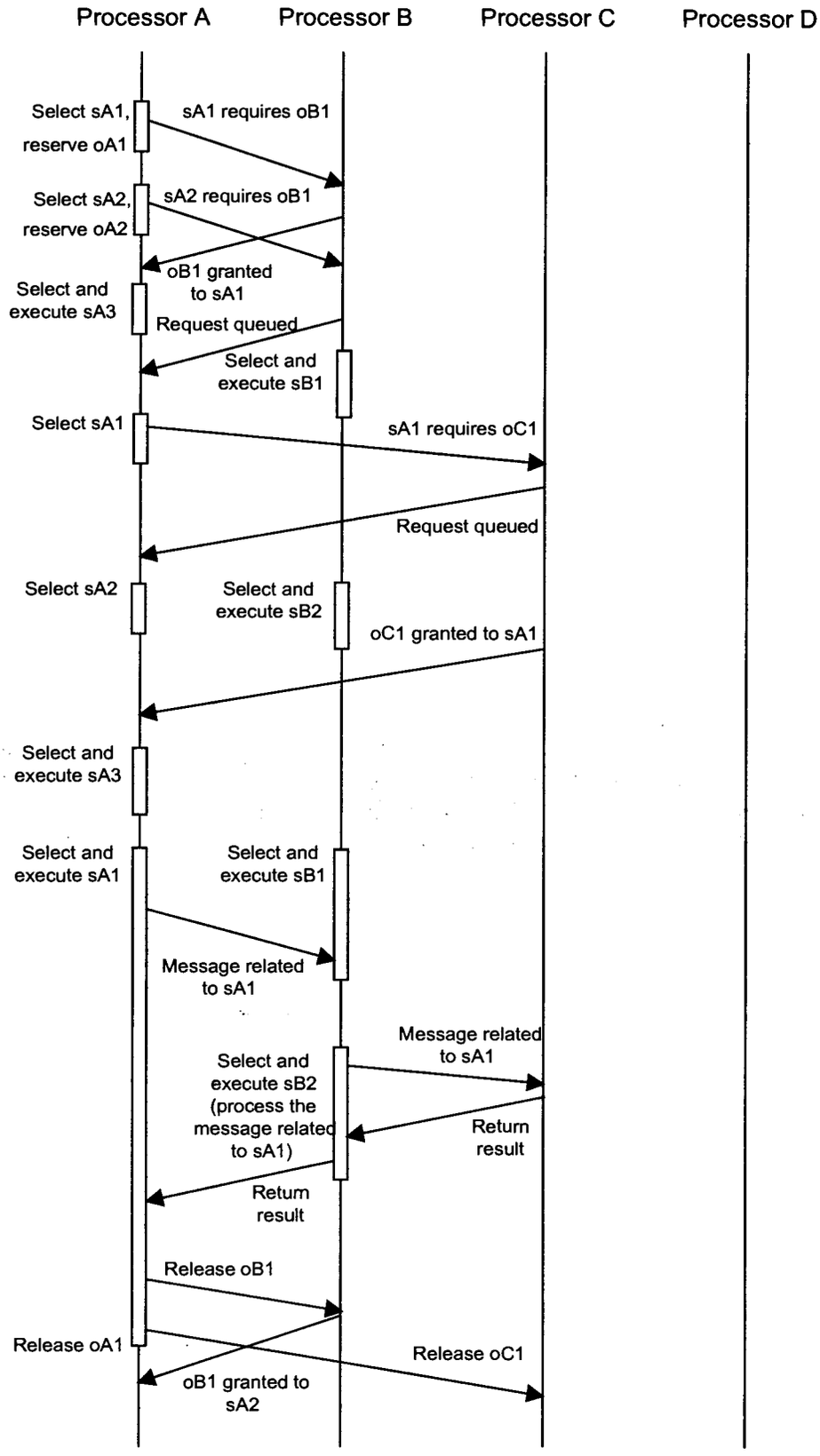

Figure 8-6. Scenario illustrating that a local parallel statement is always allowed to execute if it does not send messages to remote objects and also not to local objects that have been allocated to or requested by a remote parallel statement.

The scenario in Figure 8-7 illustrates the third condition under which a local parallel statement may be executed while some local objects are allocated to other parallel statements. In this case processor B contains a third parallel statement (*sB3*) which requires objects *oB3*, *oD1* and *oE1* and a fourth one, which requires objects *oB4* and *oC1*. When statement *sB3* is selected for execution, object *oB3* is allocated to it. A reservation request is sent to processor D in order to acquire object *oD1*. Statement *sB4* is executed next. A local reservation request is issued for object *oB4*. The request is granted, since the object is available and no local objects have been allocated to **remote** parallel statements. A request for object *oC1* is issued.

The requests for objects *oD1* and *oC1* are granted. When a reservation request for object *oB1* is received from processor A, the request is queued, because a reservation request from a remote parallel statement is always queued if any local objects are already allocated at that time. When statement *sB3* is selected again, it issues the request for object *oE1*.

Statement *sB4* is executed when it is selected, since it has acquired all its resources. It is not affected by the resource allocation status of statement *sB3*. Even if the request for *oE1* is received before statement *sB4* starts executing, statement *sB3* cannot interfere with statement *sB4*, owing to the fact that only one parallel statement is executed at a time at a particular processor and that statement execution is completed before the next statement is selected. (The messages sent to the objects located at processors C, D and E are not shown in Figure 8-7.) Once a parallel statement has completed execution, its resources are released.

The release of object *sB3* results in the allocation of object *oB1* to statement *sA1*. When statement *sB4* is selected, the local reservation request for object *oB4* is queued due to the fact that a local object (*oB1*) is allocated to a **remote** parallel statement at that time.

Figures 8-8(a) and (b) illustrate some of the consequences if these rules as implemented in the resource allocation algorithm presented in this section are not adhered to. Deadlock as a result of reservation request collision is shown in Figure 8-8(a). Processor A allocates object *oA2* to statement *sB1*, even though a local object is already allocated to statement *sA1*. This violates Rule 1. Processor B violates Rule 5 when it allocates object *oB2* to statement *sB1*. Deadlock ensues when statements *sA1* and *sB1* both start executing simultaneously.

In Figure 8-8(b) Rule 5 is violated at both processors, eventually resulting in deadlock. Figures 8-9(a) and (b) show how the rules listed above ensure correct operation under similar circumstances.

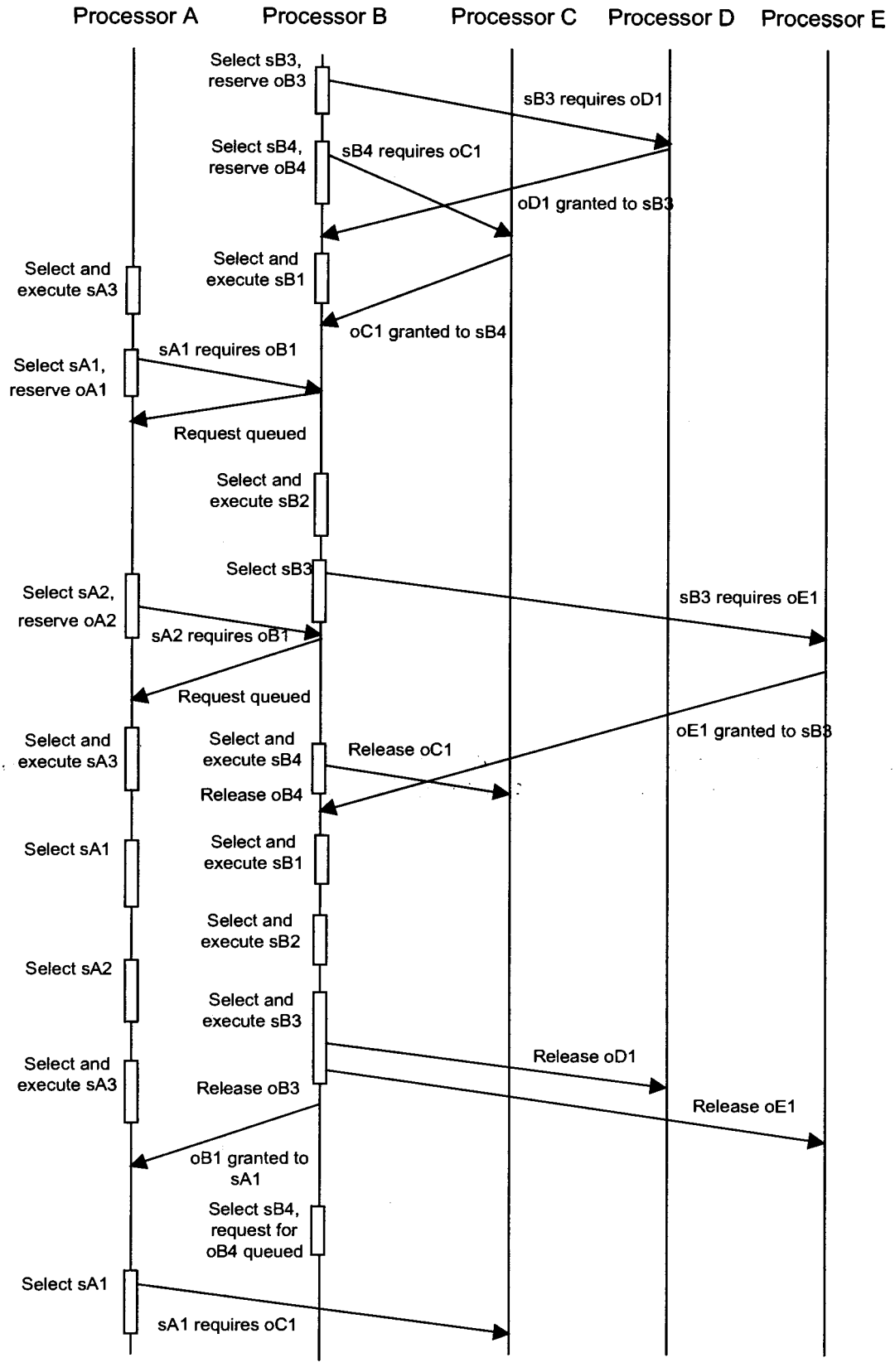

Figure 8-7. Scenario illustrating that a local parallel statement for which all resources have been granted may be executed while another local parallel statement is still waiting for remote resources to be granted. The two statements do not share local objects.

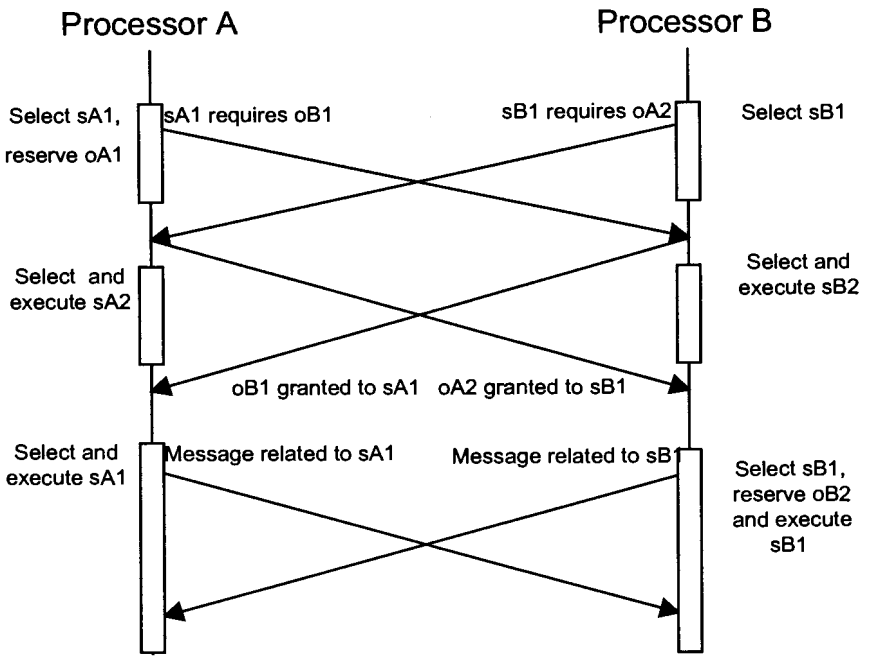

Figure 8-8(a). Deadlock in the case of reservation request collision when both local and remote reservation requests are granted at each processor.

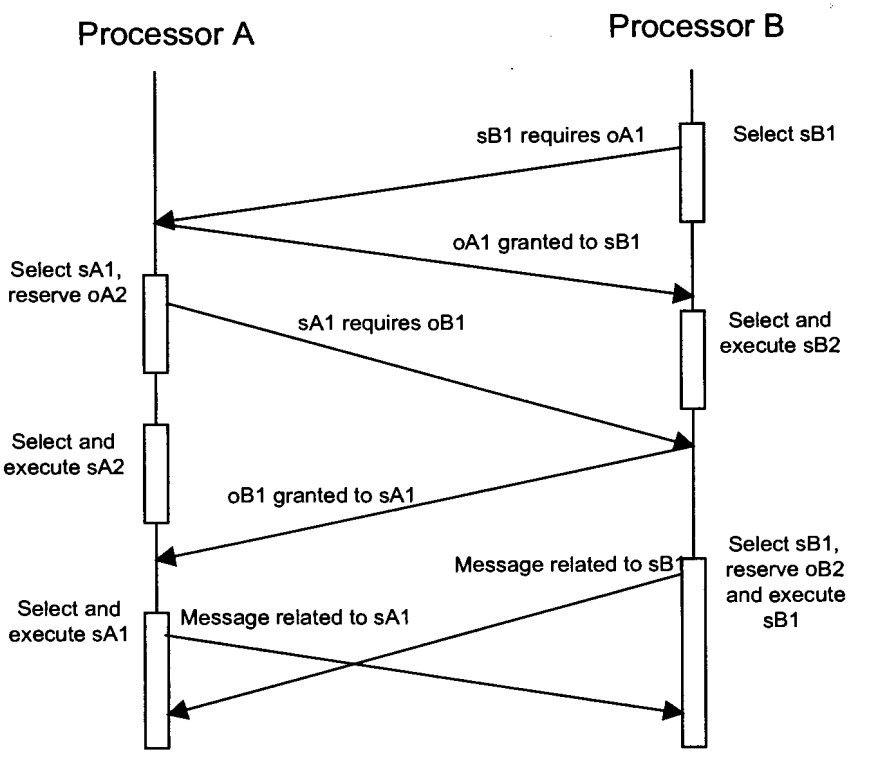

Figure 8-8(b). Deadlock when a local reservation request is granted while a local object is currently allocated to a remote parallel statement.

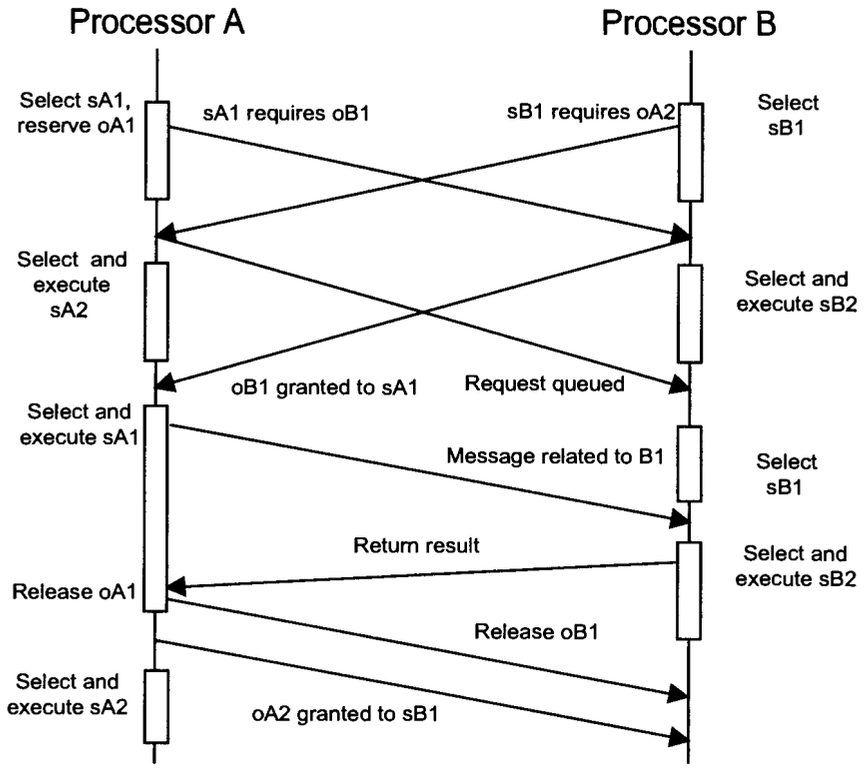

Figure 8-9(a). Deadlock prevention in the case of reservation request collision.

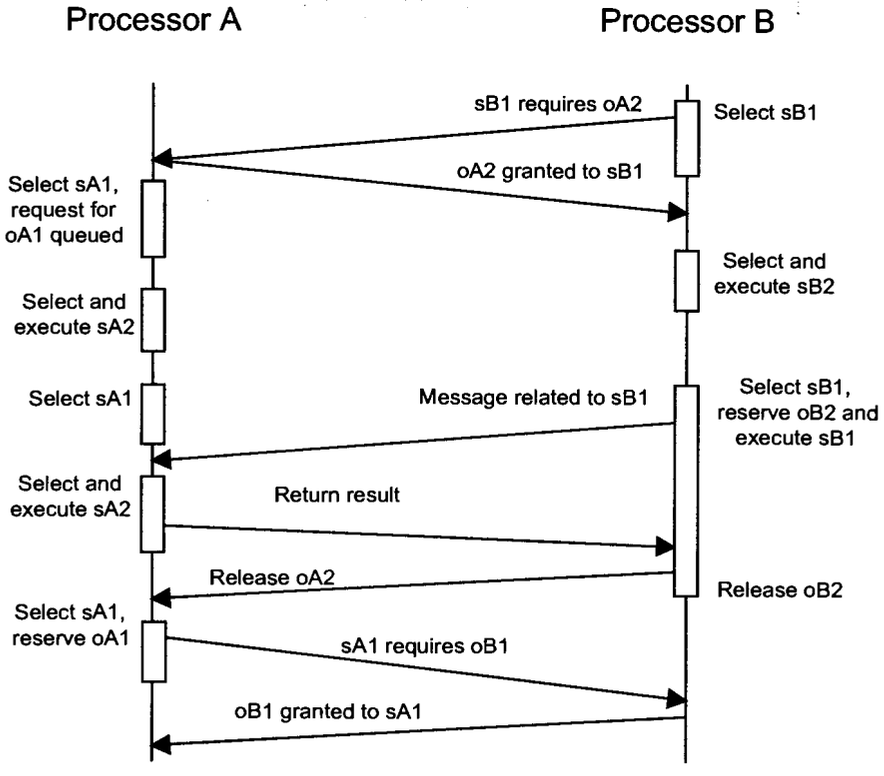

Figure 8-9(b). Deadlock prevention because a reservation request for a local object is queued while a local object has been allocated to a statement at a remote processor.

The effects of having multiple processes running on each processor:

If a distributed system comprises multiple processors, where each processor has **multiple processes** running on it, the effects on the resource allocation algorithm as described in this section are as follows:

In such a system there are multiple infinite loops running on each processor, one for each process. Each process relinquishes control of the processor in the same way as described in Section 8.3.3, i.e. the mapped SLOOP statements may not contain any messages that will yield control to the process scheduler, but after each parallel statement has been executed, control is relinquished explicitly. The atomicity of each parallel statement is thereby guaranteed.

If multiple processes may be present at each processor, only one of the processes at each processor would need to contain the statement which handles messages from remote objects. This is because the objects at that processor are shared by all the processes. It is only the parallel statements that are assigned to individual processes. In order to ensure the integrity of objects, parallel statements that share objects may not execute simultaneously.

Earlier in this section it was described how the efficiency of the resource allocation algorithm could be improved if local parallel statements could **obtain** their remote resources in **parallel**. It was stated that such concurrency would not compromise the correctness of the system, since only one parallel statement would **execute** at a time due to the fact that there was only one processor for the set of local parallel statements.

When multiple processes may run on a processor, the efficiency is improved even further, because **all** parallel statements that are **located at different processors** and that **do not share objects** could execute simultaneously. Recall that in a single process per processor distributed architecture, parallel statements can execute simultaneously if they are **located at different processors**, they **do not share objects** and the **target objects** referenced by the respective statements **do not share processors**. Since the statement which handles messages from remote objects can be assigned to a separate process if each processor has multiple processes running on it, the last condition is removed.

Reuse

It is evident from the above that the infrastructure required to map a SLOOP program to a distributed architecture is not trivial. However, once such an infrastructure has been developed, it can be reused whenever a SLOOP program needs to be mapped to such an architecture.

8.3.4.2 Identifying the objects that need to be reserved

The next issue to consider is **how to determine which objects should be reserved** for each parallel statement. This is done by inspecting the parameters used in the message expressions in the parallel statements. The receiver of the parallel statement, as well as all target objects specified as parameters have to be reserved.

If a statement only sends messages to local objects, no objects need to be reserved if all the objects can be allocated to the statement simultaneously. If a statement executes under such circumstances, it cannot interfere with any other statement, as illustrated by the examples in the previous section.

As discussed in Chapter 4, Section 4.3.5.4, parallel statements may be nested. The purpose of the parallel statements at the top nesting level at each processor is to invoke the required parallel methods of the local objects. These methods may send messages to other objects. In the SLOOP method it is a requirement that **such target objects** have to be **named explicitly as parameters**

of these methods if they do not form part of the receiver. Thus, a composite object may refer to its constituent objects without having to name them as parameters in its methods. However, any other target object has to be passed to the sending object as a parameter. When a composite object is reserved, all its components are reserved with it.

Reservation requests are only issued at the processor where the top nesting level of a parallel statement is located. Since all target objects that do not form part of the receiver of the parallel statement have to be passed as parameters to the receiver, the target objects that are involved in the execution of a parallel statement can easily be determined when the statement is selected for execution.

Due to the fact that SLOOP parallel statements may be nested, and each parallel statement may result in multiple parallel statements at the next nesting level, it is possible that all the parameters that are passed by the top level parallel statement might not be required by each parallel statement at the bottom level. This is exemplified by the parallel methods of the TimerServices[18] class.

At the top nesting level the `p_runTimer:` method is invoked with `timerEventQ` as parameter, as shown below.

```
timer p_runTimer: timerEventQ
```

The p_runTimer: method contains three parallel statements, viz.

```
    currentTime := SmalltalkLibPkg:::Time now asSeconds
[] lastTime := currentTime \+
   currentTick := (currentTick + 1) \\ (timeoutCollection size)
       if difference ≥ 1 and: [currentTimeoutElement isNil]
[] timerEventQ addLast: currentTimeoutElement \+
   currentTimeoutElement updateEndTime \+
   currentTimeoutElement timerServicesCompleted: true \+
   (timeoutCollection at: readIndex) removeFirst
       if currentTimeoutElement notNil
```

As is evident from the above, only the third statement refers to `timerEventQ`. In order to improve efficiency, the reservation requests are not made until the parallel method at the top level has been expanded to its lowest level and the relevant statement for that particular pass has been selected. This expansion procedure forms part of the mapping of a SLOOP program to a distributed architecture.

If, in the above example, the `timerEventQ` object is located at a remote processor, then no remote objects need to be reserved when the first two statements are selected for execution. These statements may be executed whenever they are selected if the `TimerServices` instance is not allocated to a remote parallel statement at the time of their selection.

The third statement requires both the `TimerServices` instance and the `timerEventQ` object to be allocated to it before it may execute. This will involve a reservation request to another processor if the `timerEventQ` object is located remotely.

Parallel messages to the pseudo-variable `self` are used for structuring purposes. It is used to invoke parallel methods that contain statements that are likely to change during subclassing. A parallel message to the variable `self` is not required to pass its own instance variables to itself as arguments. If such a message is encountered at the top nesting level it is necessary to expand the

[18] The SLOOP specification of the TimerServices class is given in Appendix B, Section B.11.

parallel statement containing this message in order to determine which objects should be reserved for the resulting statements.

For example, the `p_activate` method of the `CC_Activation`[19] class contains the following parallel statement:

```
self p_executeCPAgent
```

The `p_executeCPAgent` method of the `CC_SimulationActivation`[20] subclass contains the following parallel statements:

```
commsAgent p_simulate: timer timeoutEventsIn: timerEventQ
[] commsAgent p_generateEvent: userConnections target: inputQ
```

Thus, it is evident that the `commsAgent`, `timer` and `timerEventQ` objects are involved in the first statement, while the `commsAgent`, `userConnections` and `inputQ` objects are involved in the second statement. Since these are all instance variables of the `CC_Activation` class, it does not have to pass these parameters to itself when it sends a message to itself. It is only by expanding the statement containing the pseudo-variable `self` that it is possible to determine which objects are required for the execution of that statement .

8.3.4.3 The middleware infrastructure

Due to the advent of middleware products such as CORBA [OHE97], there is no need to be concerned about issues such as how to determine the location of an object when a message has to be sent to it. When a statement in a distributed object implementation is selected for execution, the CORBA services are used to determine the location of all the target objects involved. Although the CORBA Concurrency Control Service [OHE97] allows the designer to lock resources, this service is not used by SLOOP, since it does not provide the functionality required for the reservation of objects as described above. For example, when a lock is requested via the CORBA Concurrency Control Service, the requesting process blocks until the lock is granted, whereas the SLOOP implementation requires that it should be possible to issue reservation requests asynchronously.

Once a parallel statement starts executing, the Object Request Broker (ORB) intercepts all messages sent to a class or an instance. It takes care of locating the target object and of converting the message selector and its argument to a format that can be transmitted to a remote processor.

There are various ways of incorporating CORBA into an implementation: one option suggested in [OHE97] is to multiply inherit from the CORBA services classes. If multiple inheritance is not supported by the target architecture, CORBA allows the inclusion of *before* and *after* callbacks that are executed before and after each method respectively [OHE97]. The necessary CORBA services can be invoked from within these callbacks. The latter approach is followed in the SLOOP method, since multiple inheritance is not supported.

This concludes the discussion of the **basic** principles involved during the mapping of SLOOP programs to different types of architectures. The remainder of this chapter deals with various topics related to such mappings. For example, it is shown how the concept of reflective computation can be utilised in SLOOP mappings. There is also a discussion on how more parallelism can be introduced into a SLOOP design if that is found to be a requirement during the implementation phase. However, first more detail is given regarding the mapping of macros and different types of SLOOP statements.

---

[19] The CC_Activation class is defined in Appendix B, Section B.2.
[20] The SLOOP specification of the CC_SimulationActivation class is presented in Appendix B, Section B.3.

8.4 Mapping macros

The previous sections described how the SLOOP **computational model** can be mapped onto various architectures. However, there are other SLOOP constructs that also need consideration during the mapping process. This section covers the issues regarding the mapping of the *macros-section* of a SLOOP class or method to Smalltalk.

In Chapter 4, Section 4.4.3.3, the simplest mapping approach was presented. Each *macro-variable* is simply replaced with its corresponding *macro-expression* wherever it is used. In order to improve efficiency, other options can be considered. However, some of these options are only more efficient under very specific circumstances and could even be less efficient in others, as shown below.

The first option is as follows: For each method all the *macro-variable*s are declared as temporary Smalltalk variables. This includes the *macro-variable*s defined in the class *macros-section* that are referenced by the specified method, as well as those in the *macros-section* of the specified method. If a *macro-expression* contains another *macro-variable*, the latter has to be declared as a temporary variable as well. The *macro-expression*s are evaluated and the results are assigned to the corresponding *macro-variable*s when a parallel or sequential method is **entered**.

This approach is more efficient than the mapping given in Chapter 4, Section 4.4.3.3, if the same *macro-variable* is referenced multiple times in a sequential method or if it is referenced multiple times within the same statement in a parallel method. However, this mapping only produces the correct results if the *macro-expression* has the same value at all locations in the method. In the case of a parallel method, where only one statement is selected for execution at each invocation, the *macro-variable*s may not even be referenced in the selected statement. In that case the evaluation of the *macro-expression* is a wasted computation.

Another option is to use a **hybrid** approach, i.e. in **sequential** methods macros are evaluated **once** when the method is **entered**, while in **parallel** methods the *macro-variable*s are replaced with their corresponding *macro-expression*s **wherever they are used**. Again this solution can only be used if each *macro-expression* always has the same value regardless of the location in the sequential method. Thus, if the *macro-variable* has a different value at different occurrences within a sequential method, a temporary variable mapping will not preserve the semantics of the SLOOP statements.

The simplification of the implementation of correctness property checks is another argument in favour of merely replacing *macro-variable*s with their corresponding *macro-expression*s wherever they are used. If the *properties-section* is implemented using reflection (as discussed in Section 8.6.3), then the *macro-expression*s in the property specifications are evaluated from within the metaclass by obtaining the relevant values of class and instance variables from the base class. The alternative options above map the *macro-variable*s to temporary variables. The scope of the temporary variables is the method within which they appear, i.e. at the time when the **preconditions** are checked in the metaclass no values have been assigned to these temporary variables yet. However, since it is **optional** to implement the *properties-section*s of a SLOOP program, this will not be a consideration if the *properties-section*s are not mapped to the executable program.

Thus, the simplest mapping of *macro-variable*s would be to replace them with their *macro-expression*s wherever they are used. In order to improve the efficiency of the algorithm, temporary variables could be used in **some** situations, but the designer would have to take care that the **semantics** of the SLOOP statement are preserved.

8.5 Mapping SLOOP statements

The mapping of a SLOOP statement which contains a single *statement-component* and which, in turn, contains a single *component-part* is straightforward as was demonstrated in Chapter 4, Section 4.4.3.3. This section deals with the Smalltalk mapping of parallel methods that contain multiple *quantified-statement-lists*. It also covers the mapping of statements comprising multiple *statement-components* and *component-parts*.

8.5.1 Mapping quantified-statement-lists

When multiple *quantified-statement-lists* appear in a **parallel** method, each statement within each quantification has to be included in the list of statements that are executed infinitely often. It has to be included in such a way that only one statement is selected at each invocation of the method. In the code fragment below, two of the *quantified-statement-lists* appearing in the `p_activate` method of the `CC_Activation class`[21] are shown.

```
[] < [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
        >
[] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
      p_execute
        >
```

When the class is instantiated, the `p_activateTally` instance variable is set to the total number of parallel statements contained within the `p_activate` method. For brevity some of the statements in the `p_activate` method are omitted in this example. The mapping below only shows the statements above, as well as two other enumerated statements.

The `p_activateTally` variable is set to `2 + (config maximumServiceCategories) + (config maximumConnections)` in the `initialize` method. In the `p_activate` method, the first statement calculates which parallel statement should be executed. The second statement calculates which of the quantified statements should be executed, should the current value of `p_activateCycleIndex` not indicate one of the enumerated statements. The temporary variables `i` and `j` are used to select the correct instances of the `Connection` and `ServiceCategory` classes respectively.

```
"Determine which statement should be executed."
p_activateCycleIndex :=
      ((p_activateCycleIndex + 1) \\ p_activateTally).

"Determine whether it is one of the statements of the
Connection class"
(p_activateCycleIndex > 1 and:
[p_activateCycleIndex ≤ (1 + config maximumConnections)])
      ifTrue: [i := i+1]
      ifFalse:
      [
            i := 0.
            "Determine whether it is one of the statements of
            the ServiceCategory class"
            (p_activateCycleIndex >
            (1 + config maximumConnections) and:
            [p_activateCycleIndex ≤
```

[21] The SLOOP specification of the CC_Activation class is presented in Appendix B, Section B.2.

```
                (1 + config maximumConnections +
                config maximumServiceCategories)])
                      ifTrue:  [j := j+1]
                      ifFalse: [j := 0].

(p_activateCycleIndex = 0)
ifTrue: [self p_executeCPAgent]                    "statement 0"

ifFalse: [(p_activateCycleIndex = 1)
      ifTrue: [timer p_runTimer: timerEventQ]      "statement 1"

      ifFalse: [ (p_activateCycleIndex = (1+i))
            ifTrue: [self p_executeConnection:
                  (userConnections at: i)]      "statement 1+i"

            ifFalse: [ (p_activateCycleIndex = (1 +
                        config maximumConnections + j)
                  ifTrue: [(scContainer at: j) p_execute ]
                  "statement 1 + config maximumConnections + j"
                  ifFalse:[...
                        ]
                  ]
            ]
      ]
```

The above code fragment represents one possible way of mapping *quantified-statement-lists*. Many other mappings are possible, but the above suffices as an illustration of how it could be done.

8.5.2 Mapping statement-components and component-parts

The next example illustrates the mapping of a SLOOP statement comprising multiple *statement-components* and *component-parts*. This statement from the `p_generateEvent:target:` method of the `CommsProviderSimulator` class[22] has two *statement-components* separated by the '||' symbol. (For the purposes of this example, the two *statement-components* appear in a specific order in this section, which happens to be different from their ordering in Appendix B. This is acceptable, since the order in which *statement-components* appear in a SLOOP program is not significant.)

```
newEventRequired := true \+
generatingEvent := false
      if generatingEvent
||inputQ addLast: (idleConnection serviceRequest) \+
idleConnection assign
      if generatingEvent and: [idleConnection notNil] ~
Transcript show: 'All connections busy'
      if generatingEvent and: [idleConnection isNil]
```

(The *macro-variable* `idleConnection` is defined as `self getIdleConnection: userConnections`.)

It is important that the semantics of the statement should be preserved, otherwise the resulting program could be incorrect. The evaluation order of a SLOOP statement was first presented in Chapter 4, Section 4.3.6.3. Recall that all conditional expressions appearing in a statement are evaluated first. Thereafter the message expressions that represent receivers and arguments are

[22] The CommsProviderSimulator class is specified in Appendix B, Section B.6.

evaluated. Finally, the resulting message expressions (the only ones that may modify variables) are evaluated.

If, instead of following these rules, the statement above is mapped to a sequential architecture by executing the *statement-component*s sequentially, `generatingEvent` would be set to false before the conditional expressions of the second *statement-component* are evaluated. No service requests would ever be added to the `inputQ`, even if there are idle connections, as can be seen below.

```
(generatingEvent)
ifTrue:
[
      newEventRequired := true.
      generatingEvent := false
].
(generatingEvent and:[(self getIdleConnection: userConnections)
 notNil])
ifTrue:
[
      inputQ addLast:
      (self getIdleConnection: userConnections) serviceRequest.
      (self getIdleConnection: userConnections) assign
]
ifFalse: [Transcript show:'All connections busy'].
```

A correct mapping of the above statement for a sequential architecture is the following:

```
(generatingEvent)
ifTrue:
[
      newEventRequired := true.
      generatingEvent := false.
      ((self getIdleConnection: userConnections) notNil)
      ifTrue:
      [
            inputQ addLast:
            (self getIdleConnection: userConnections) serviceRequest.
            (self getIdleConnection: userConnections) assign
      ]
      ifFalse: [Transcript show:'All connections busy'].
]
```

There are several correct mappings for the above statement. The important issue is to make sure that if the value of a variable is **used** and **modified** in the **same** statement, then its value **prior** to the modification has to be obtained for all occurrences where it is **used**. All of these occurrences have to use the value obtained **prior** to the modification.

A modification may be performed explicitly via an assignment that forms part of the parallel statement, or it may occur implicitly via an assignment that is performed as a result of a method being invoked from within the parallel statement. When `generatingEvent` is set to false in the above statement, it is an example of an explicit modification. The invocation of the `assign` method of the `Connection` class in the above statement is an example of an implicit modification.

Note that in a synchronous shared-memory architecture as described in Sections 8.2.2 and 8.3.2, each *component-part* of a parallel statement can be assigned to a different processor. Thus, the `addLast:` and the `assign` methods in the above statement can be executed simultaneously, and at the same time the values of `newEventRequired` and `generatingEvent` can also be

updated. The various *statement-component*s and *component-part*s of a parallel statement may therefore be executed concurrently, with the proviso that **all** read accesses performed by any *statement-component* or *component-part* of the statement are completed before any write access may commence.

8.6 The use of reflection in mappings of SLOOP programs

In Chapter 1, Section 1.3.4, it was stated that computational reflection could be used, inter alia, to **reason about control** and for **assertion checking**. These topics are now explored further.

Each time when a parallel method is invoked, only one of its constituent statements is executed. In the mapping to an executable Smalltalk program additional variables and logic are introduced in order to select a statement for each invocation. In Section 8.6.2 it is shown how this aspect of the mapping can be delegated to the metaclass of each base class, thereby ensuring that the latter is not cluttered with variables and statements that are irrelevant to the class itself.

When a method from the base class is invoked, it is possible to check the preconditions for that method in the metaclass before the method is executed. Once the method has completed, the postconditions can be checked in the metaclass before control is returned to the client. This aspect of the use of reflection in the SLOOP method is covered in Section 8.6.3. However, first more information is given regarding the infrastructure that is required in order to make use of reflection in this way.

8.6.1 A reflective computation infrastructure

The ALBEDO meta-object infrastructure [Bekk93] is one possible infrastructure that can be used to provide reflective computational facilities. The basic principle employed in a meta-object infrastructure is to associate a metaclass with a base class and to allow the metaclass object to **intercept** the messages sent to the base class object.

The ALBEDO meta-object infrastructure provides a **mechanism** to intercept these messages. It is implemented in Smalltalk-80. The Smalltalk architecture consists of a virtual machine and a virtual image [GoRo89]. The virtual machine handles the interface to the hardware and also contains low level routines that must be written in machine language. The virtual image consists of the kernel objects, the compiler objects and the users' objects. Since the **message handling** primitive of Smalltalk is part of the virtual machine, it is inaccessible to users.

The way in which this problem is overcome in the ALBEDO system is by **encapsulating** the base object and its associated meta-object in a **shell**. A minimal set of methods is implemented for the class representing this shell. As a result most messages received by the shell are not understood. The `doesNotUnderstand:` method is reimplemented in the shell. Instead of displaying an error message, the `handleMsg: aMessage` message is sent to the encapsulated meta-object, where `aMessage` is the original message received by the shell.

The meta-object takes the necessary actions pertaining to that particular metaclass and it then passes the original message to its associated base object. Depending on the function of the meta-object, it may even modify the original message before sending it to the base object. This aspect will be discussed in more detail below. A graphical representation of the architecture of the infrastructure is presented in Figure 8-10.

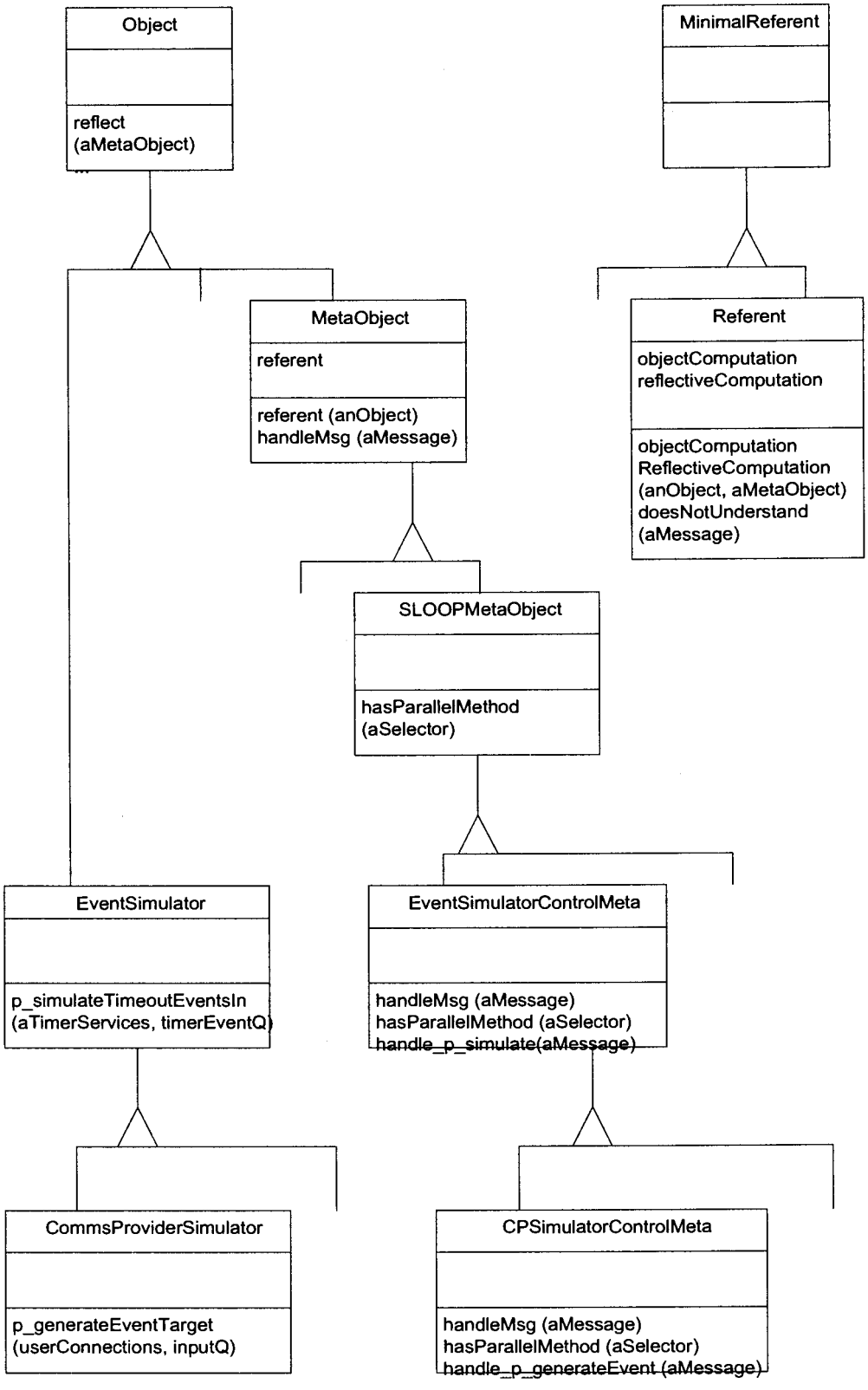

Figure 8-10. Class diagram of the ALBEDO meta-object infrastructure based on Smalltalk.

Only the most important instance variables and methods relevant to the discussion are shown in the diagram. The `EventSimulator` and `CommsProviderSimulator` classes[23] are used as examples of a base class hierarchy. The `Object` class is their root class. To a large extent the metaclass hierarchy mimics the base class hierarchy. However, all metaclasses have to be descendants of the `MetaObject` class. The latter is a subclass of `Object`. It is also not mandatory for each base class to have a metaclass associated with it. The purpose of the `SLOOPMetaObject` class is to add methods and instance variables that are specific to SLOOP mappings. This class and its descendants are described in more detail later on in this section.

The **implementation of the shell** is realized by two new classes, viz. `MinimalReferent` and `Referent`. `MinimalReferent` is a new root class. A minimal set of methods from the Smalltalk `Object` root class is recompiled for `MinimalReferent`. The subclass `Referent` contains the functionality specific to the implementation of reflective computation. This class contains the two instance variables `objectComputation` and `reflectiveComputation`. Once the encapsulation has been performed via the new `reflect:` method of the `Object` class, they refer to the base object and meta-object respectively. The `reflect:` method will be discussed in more detail shortly. The `Referent` class also contains the reimplemented `doesNotUnderstand:` method. Reflective computation is activated as illustrated by the example of the `CommsProviderSimulator` class given next.

The `initialize` method of the `CC_Activation` class contains the following statement:

```
commsAgent := self initCommsAgent
```

The `initCommsAgent` method is defined as being the responsibility of the subclass (in this case `CC_SimulationActivation`). In the original `CC_SimulationActivation` class this method contains the following statement:

```
^CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation
```

The `initCommsAgent` method is now augmented with the statements related to reflective computation (the additions are shown in bold):

```
|commsAgentBase commsAgentMeta|
commsAgentBase :=
      CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation.
commsAgentMeta :=
      CC_SimulationInterfacesPkg:::CPSimulatorControlMeta new.
commsAgentBase := commsAgentBase reflect: commsAgentMeta.
^commsAgentBase
```

The `reflect:` method is a **new method added to the `Object` class** by the ALBEDO meta-object infrastructure. Its purpose is to **encapsulate** the base object and its meta-object in a shell. The implementation of the `reflect:` method is as follows:

```
"Statement(s) of the reflect: method"
^Referent objectComputation: self reflectiveComputation: aMetaObject
```

Thus, the `objectComputation:reflectiveComputation:` class method of the `Referent` class is invoked. This method creates and returns a new instance of `Referent` and it causes the instance variables `objectComputation` and `reflectiveComputation` to refer to the encapsulated base object and meta-object respectively. At this stage the `referent` instance variable of `commsAgentMeta`, the meta-object, is also set to refer to `commsAgentBase`, its

[23] The EventSimulator and CommsProviderSimulator classes are specified in detail in Appendix B, Sections B.5 and B.6 respectively.

associated base object. (The `CPSimulatorControlMeta` class inherits the `referent` instance variable from the `MetaObject` class.) The base object has no reference to its encapsulation or its associated meta-object.

In this example the `commsAgentMeta` meta-object is an instance of the `CPSimulatorControlMeta` class. This class performs reflective computation related to the **control** of parallel statements in the base class. It is possible to instantiate a different type of metaclass at this point. For example, a metaclass that performs reflective computation related to assertion checking or one that performs both control actions and assertion checking could be instantiated. In the remainder of this section the general concepts are explained using the control metaclasses as examples.

In the program fragment above the `commsAgentBase` variable is set to the value returned by the `reflect:` method. This is also the value that is returned by the modified version of the `initCommsAgent` method. Thus, instead of referring to a `CommsProviderSimulator` instance, the `commsAgent` instance variable refers to the **shell** encapsulating the `CommsProviderSimulator` instance and its associated meta-object. (The shell is an instance of `Referent`.)

When a message is now sent to `commsAgent`, the **shell** receives it. Since the shell understands only a minimal set of messages, the `doesNotUnderstand:` method is invoked for most messages. The `doesNotUnderstand:` implementation for the `Referent` class is as follows:

```
^reflectiveComputation handleMsg: aMessage.
```

Thus, the message is passed to `commsAgentMeta`, the meta-object. The `handleMsg:` method now performs the reflective computation before it passes the message to the base object.

(The `selector` and `respondsTo` : methods invoked in the program fragment below refer to Smalltalk-80 library methods and are not shown in Figure 8-10. The receiver of the `selector` method is a message expression. The `selector` method returns the selector found in this message expression. For example, the `(inputQ addLast: serviceRequest) selector` message expression would return the `addLast:` selector. Thus, if `(inputQ addLast: serviceRequest)` is passed as argument to the `handleMsg:` method below, then `aMessage` would have the value `(inputQ addLast: serviceRequest)` and `msgSelector` would have the value `addLast:`. The `respondsTo:` method returns true if the receiver supports the method specified as argument of the `respondsTo:` method.)

The `handleMsg:` method of `CPSimulatorControlMeta` is as follows:

```
|msgSelector|
msgSelector := aMessage selector.
(referent respondsTo: msgSelector)
ifTrue:
[
        (msgSelector = #p_generateEvent:target:)
        ifTrue:
                [^self handle_p_generateEvent: aMessage]
        ifFalse:
        [
                (super hasParallelMethod: msgSelector)
                ifTrue: [^super handleMsg: aMessage]
                ifFalse:
                [^referent perform: msgSelector
                withArguments: (aMessage arguments)]
        ]
]
```

```
ifFalse:
        [^referent doesNotUnderstand: aMessage]
```

First of all, the meta-object checks whether the base-object can respond to the specified message selector. If it cannot, the `doesNotUnderstand:` method associated with the base-object is invoked (i.e. the original one implemented in the `Object` class, which displays an error message).

If the message can be understood, the meta-object performs its reflective computation. In the case of a `CPSimulatorControlMeta` instance, which needs to take control actions, the meta-object then checks whether it indicates one of the parallel methods of the base class (in this case there is only one, viz. `p_generateEvent:target:`).

If it does, `handle_p_generateEvent:`, the meta-object method which handles the selected parallel method, is invoked. The contents of `handle_p_generateEvent:` is discussed in Section 8.6.2. At this stage it suffices to note that the method selects one of the parallel statements of the `p_generateEvent:target:` method and ensures that the selected statement is executed.

If the message selector does not match one of those supported by the base-object, the superclass of the current meta-object is consulted. If the superclass (or one of its ancestors) can find a match for the specified message selector, the message is passed to the relevant ancestor. If no match can be found, it is assumed that the message selector represents a sequential method and the message is passed to the base-object using explicit message passing.

The `CommsProviderSimulator` class inherits the `p_simulate:timeoutEventsIn:` method from `EventSimulator`, its parent class. The reflective computation related to this method is found in the corresponding metaclass. Details regarding the `handle_p_simulate:` method will be given in Section 8.6.2.

The `handleMsg:` method of `EventSimulatorControlMeta` is as follows:

```
|msgSelector|
msgSelector := aMessage selector.
(referent respondsTo: msgSelector)
ifTrue:
[
        (msgSelector = #p_simulate:timeoutEventsIn:)
        ifTrue:
                [^self handle_p_simulate: aMessage]
        ifFalse:
        [
                (super hasParallelMethod: msgSelector)
                ifTrue:
                        [^super handleMsg: aMessage]
                ifFalse:
                [^referent perform: msgSelector
                withArguments: (aMessage arguments)]
        ]
]
ifFalse:
        [^referent doesNotUnderstand: aMessage]
```

The class `MetaObject` is the superclass of all meta-classes. The subclass `SLOOPMetaObject` merely adds the method which checks whether the associated base class contains any parallel methods. In `SLOOPMetaObject` the method is implemented as follows:

```
hasParallelMethod: aSelector
^false
```

Each subclass of `SLOOPMetaObject` which performs reflective computation about the parallel statements in its associated base class has to reimplement this method. Thus, in `EventSimulatorControlMeta` (subclass of `SLOOPMetaObject`) it is reimplemented as shown below:

```
hasParallelMethod: aSelector
(aSelector = #p_simulate:timeoutEventsIn:)
ifTrue: [^true]
ifFalse: [^super hasParallelMethod: aSelector]
```

`CPSimulatorControlMeta` (subclass of `EventSimulatorControlMeta`) also reimplements the method:

```
hasParallelMethod: aSelector
(aSelector = #p_generateEvent:target:)
ifTrue: [^true]
ifFalse: [^super hasParallelMethod]
```

In the next section it will be shown how reflective computation is used to **select a parallel statement** for execution.

8.6.2 Using reflective computation for control purposes

In Section 8.3.1 it was described how the selection of a single parallel statement per parallel method invocation can be achieved on a sequential architecture. The mechanism is based on the introduction of two additional variables. The one contains the total number of statements within the method and the other variable is used to record which statement had been executed during the previous cycle.

Instead of adding these variables to the base classes, they are added to the meta-classes. Each meta-object has to initialize these variables. If such variables are inherited from a superclass, it has to be ensured that the variables in the superclass are initialized as well. This is illustrated by the example below. The `new` method of the `EventSimulatorControlMeta` class is implemented as follows:

```
^(super new) eventSimulatorInit
```

The `eventSimulatorInit` method initializes the `p_simulateTally` and `p_simulateCycleIndex` variables:

```
"Statements of the eventSimulatorInit method"
p_simulateTally := 2.
p_simulateCycleIndex := p_simulateTally - 1.
```

The `CPSimulatorControlMeta` metaclass is a subclass of `EventSimulatorControlMeta`. Thus, when this class is instantiated, it has to ensure that the instance variables inherited from its superclass are initialized as well. Its new method is therefore implemented as follows:

```
^(super new) cpSimulatorInit
```

Thus, first of all the `new` method of `EventSimulatorControlMeta`, its superclass, is invoked. That results in an instance being created, as well as in the invocation of the

`eventSimulatorInit` method. The instance variables of its superclass are therefore initialized. Subsequently the `cpSimulatorInit` method is invoked. The latter initializes the instance variables of the `CPSimulatorControlMeta` class, as shown below:

```
"Statements of the cpSimulatorInit method"
p_generateEventTally := 1.
p_generateEventCycleIndex := p_generateEventTally - 1.
```

When a message is received by the shell and it has been passed to the encapsulated meta-object via the `handleMsg:` method, the meta-object determines whether the message selector indicates a parallel method or a sequential method. It also determines whether it is a parallel method supported at the current level in the class hierarchy or by an ancestor. If it is a parallel method selector, the relevant meta-object method is then invoked to perform the selection of the appropriate statement. This procedure was covered in the previous section. The implementation of the meta-object methods that perform the statement selection is now described.

The purpose of moving the parallel statement selection (i.e. **control**) functionality to the metaclass level is to keep the mapping of the base class as close as possible to the original SLOOP statements. However, there is one aspect that cannot be kept transparent to the base class, viz. the fact that it has to be possible to refer to a single parallel statement at a time.

If a parallel method contains multiple statements (which may be convenient for abstraction purposes), the Smalltalk mapping has to include a separate method for each parallel statement within the method. These methods are not visible to any other base classes. Thus, all other base classes refer to the parallel method that corresponds with the SLOOP parallel method. The methods containing the single parallel statements are only used by the corresponding metaclass. This is now exemplified by the `EventSimulator` and `CommsProviderSimulator` classes.

The SLOOP version of the statements in the `p_simulate:timeoutEventsIn:` method is shown first. It contains two parallel statements:

```
self startRandomTimer: aTimerServices withMaximum:
(aTimerServices maximumTimeout) \+
newEventRequired := false
      if newEventRequired
[] generatingEvent := true \+
self resetTimerExpired: timerEventQ
      if self timerExpired: timerEventQ
```

The `handle_p_simulate:` aMessage method of the `EventSimulatorControlMeta` class is implemented as follows:

```
handle_p_simulate: aMessage

p_simulateCycleIndex := (p_simulateCycleIndex + 1) \\ p_simulateTally.
args := aMessage arguments.

(p_simulateCycleIndex = 0)
if True:
[
      ^referent perform: (#p_s1_simulate:) withArguments: (args at: 1)
]
ifFalse:
[
      (p_simulateCycleIndex = 1)
      ifTrue:
            [^referent perform: (#p_s2_simulate:)
```

```
            withArguments: (args at: 2)]
]
```

The corresponding single statement methods in the base class are as follows:

```
p_s1_simulate: aTimerServices

(newEventRequired) ifTrue:
[
      self startRandomTimer: aTimerServices withMaximum:
      (aTimerServices maximumTimeout).
      newEventRequired := false
]


p_s2_simulate: timerEventQ

(self timerExpired: timerEventQ) ifTrue:
[
      generatingEvent := true.
      self resetTimerExpired: timerEventQ
]
```

Note that the original `p_simulate:timeoutEventsIn:` method contained two arguments. However, each of its constituent parallel statements only refers to one of these arguments. This is reflected by the new methods invoked by the metaclass.

The `handle_p_generateEvent:` aMessage method of the `CPSimulatorControlMeta` class is implemented as follows:

```
handle_p_generateEvent: aMessage

p_generateEventCycleIndex :=
      (p_generateEventCycleIndex + 1) \\ p_generateEventTally.
args := aMessage arguments.

(p_generateEventCycleIndex = 0)
if True:
      [^referent perform: (aMessage selector): withArguments: args]
```

Note that in this case the parallel method in the SLOOP class contains only one statement as shown below, in which case there is no need to define additional single statement methods.

```
      inputQ addLast: (idleConnection serviceRequest) \+
      idleConnection assign
            if generatingEvent and: [idleConnection notNil] ~
      Transcript show: 'All connections busy'
            if generatingEvent and: [idleConnection isNil]
      || newEventRequired := true \+
      generatingEvent := false
            if generatingEvent
```

The ALBEDO meta-object infrastructure does have the limitation that messages to the Smalltalk pseudo-variables `self` and `super` are not intercepted [Bekk93]. The reason for sending a parallel message to `self` is purely for structuring purposes. Equivalent functionality is achieved by replacing the statement containing the message to `self` with the constituent statements of the corresponding method. Thus far there has been no requirement for sending parallel messages to

`super`. An example of parallel statements containing messages to `self` can be found in the `p_activate` method of the `CC_Activation` class in Appendix B, Section B.2. This concludes the discussion on how reflective computation can be used to control which parallel statement should be executed next. In the next section another application of reflective computation is covered, viz. assertion checking.

8.6.3 Using reflective computation for assertion checking

It is of the utmost importance to check that the relevant correctness properties are not violated by the mapping of any SLOOP program fragment to an executable program. Some of these properties can be checked at run-time using the reflective facilities of Smalltalk. In particular, each message can be intercepted by the meta-object of the target object. The preconditions are checked before the message is passed to the target object. When the latter has completed its execution of the corresponding method, control returns to the meta-object, which then checks the postconditions before returning control to the client object. The ALBEDO meta-object infrastructure [Bekk93] as described in Section 8.6.1 can again be used to achieve this.

However, the **liveness** and **precedence** properties of a parallel method cannot be checked in this way. The postconditions of these properties are only required to hold eventually, provided the preconditions hold at some point and the method is invoked infinitely often. During the implementation phase it is therefore necessary to ensure that each parallel statement will indeed be executed infinitely often. In Section 8.2 it was described how this could be guaranteed by enclosing the relevant statements in infinite loops. It is imperative to check that the variables representing the total number of statements in each parallel method contain the correct values. The algorithm used to select the next statement within a parallel method has to guarantee that each statement within the method will eventually be selected. The software designer therefore has to take special care that these aspects of the correctness of the mapping are thoroughly checked.

Apart from reasoning about the correctness of the mapping, it is also possible to implement trace statements as part of the meta-objects. In so doing, the statements of the base objects are not affected, while an additional mechanism is provided to increase confidence in the correctness of the mapping.

Computational reflection is therefore a powerful mechanism that can be used during the implementation phase of a project based on the SLOOP method. It facilitates the separation of concerns, i.e. the base classes represent the SLOOP classes, while the metaclasses contain the information about those classes that are implicitly present in the SLOOP classes.

8.7 Modifying the level of parallelism in a SLOOP design

In some cases, depending on the target architecture to which the SLOOP program should be mapped, it could be found during the implementation phase that more parallelism in the design would be beneficial. The SLOOP approach makes it relatively easy to introduce more parallelism into a design.

For example, when the `ServiceCategoryAllocator`[24] instance needs to categorise the service request at the head of the `inputQ`, it does this by executing the following parallel statement:

```
categorising := true \+
self categoriseServiceRequest: (inputQ first) using: scContainer
      if inputQ isEmpty not and: [categorising not]
```

[24] The ServiceCategoryAllocator class is specified in Appendix B, Section B.8.

The purpose of the `categorising` instance variable is to ensure that the `categoriseServiceRequest:using:` method is only invoked once for the service request at the head of the `inputQ`. Once the service request has been categorised, another parallel statement[25] is executed which removes it from the `inputQ` and at the same time sets the `categorising` variable to false, thereby facilitating the `categorisation` of the next element in the `inputQ`. Depending on the requirements of the system, the categorisation of a service request could be lengthy and complex.

One disadvantage of the formulation of the above statement is the fact that in order to categorise an entry from the `inputQ`, all the objects that are involved at various stages of the categorisation have to be reserved. Thus, `scAllocator`, `inputQ` and all its elements, as well as `scContainer` and all its elements have to reserved before this statement can be executed.

It is desirable to reduce the number of objects that have to be reserved for a particular statement, since that would decrease the period for which those objects are tied up. This is because they are not released until all of the relevant objects have been reserved and the statement has completed its execution.

A higher degree of parallelism is achieved by splitting the statement shown above into two statements, simply by introducing an additional instance variable, viz. `currentServiceRequest`. The resulting statements are shown next:

```
        categorising := true \+
        currentServiceRequest := inputQ first
            if inputQ isEmpty not and: [categorising not]
    [] self categoriseServiceRequest: currentServiceRequest
       using: scContainer
            if currentServiceRequest notNil
```

The `currentServiceRequest` instance variable is set to nil inside the `categoriseServiceRequest:using:` method.

The above design facilitates a higher level of parallelism. For example, the `CommsProviderSimulator` class statement which adds new service requests to the `inputQ` and the `ServiceCategoryAllocator` statement which invokes the `categorise=ServiceRequest:using:` method can be executed concurrently (provided these two statements or the objects that they refer to do not share processors). It is therefore quite clear that it is relatively simple to introduce more parallelism into a design. The only issue that has to be taken into account whenever such modifications are made, is the fact that none of the correctness properties should be violated by the modifications.

8.8 Summary

This chapter covered various aspects related to the implementation phase of the SLOOP method.

In earlier chapters it was often stated that the SLOOP method encouraged a **unified** approach towards system design; there was no need to consider the target architecture during the analysis and design phases. This chapter served to reaffirm this view. It demonstrated that a **single** SLOOP program could be mapped successfully to **sequential**, **synchronous shared-memory**, **asynchronous shared-memory** and **distributed** architectures.

[25] This second parallel statement is not shown here, but can be found in the p_allocate:from: method of the ServiceCategoryAllocator class specified in Appendix B, Section B.8

An important aspect of the SLOOP method which emerged from the discussion of the mapping to distributed architectures is the **high level of abstraction** of the SLOOP program. The designer may rely on the **atomicity** of each parallel statement when issues such as exclusive access to objects and the execution of critical sections have to be considered. The **mapping** to the executable program has to ensure this atomicity. In Section 8.3.4.1 it became apparent how much of the **complexity** of the total system was **delegated** to the supporting infrastructure when it was shown briefly **how** the atomicity could be guaranteed in a distributed system environment. A **further level of delegation of functionality** is the use of a **middleware product** such as CORBA to take care of issues such as the converting of the message selector and its argument to a format that can be transmitted to a remote processor.

Another advantage of the **separation of concerns** is the fact that the supporting infrastructure only needs to be developed once. Thereafter it can be **reused** by any SLOOP program. This greatly simplifies correctness reasoning. The designer using the SLOOP method only needs to consider the SLOOP statements. The atomicity of these statements can be relied upon, since the behaviour of the supporting infrastructure can be assumed to be correct once it has been proved. The correctness properties of the supporting infrastructure are therefore being reused by the designers of the SLOOP programs.

Ideally, a SLOOP development environment should include the required supporting infrastructures for various architectures. Building such an environment is one of the subjects for future research.

Other aspects of the mapping of a SLOOP program to an executable Smalltalk program include the mapping of the *macros-sections* and the various types of SLOOP statements. These discussions were included to demonstrate that it was possible to perform these mappings with **relative ease**. A SLOOP translator could automate all of these tasks. The design and implementation of such a translator to a given target language (e.g. Smalltalk) is another subject for further study.

As a step towards making the final executable program look as much as possible like the original SLOOP program, it was shown that the **reflective facilities** of Smalltalk could be used to **select the next parallel statement** for execution. Again, this functionality could form part of the development environment. Similarly, reflective computation could also be used to perform **assertion checking**.

In addition to emphasizing the high level of abstraction of a SLOOP program, the relative ease with which more **parallelism** could be introduced into a SLOOP program was also pointed out.

Throughout this chapter the importance of ensuring the **adherence to the specified correctness properties throughout the development life cycle** was stressed. Some measures that may be taken to avoid the violation of these properties during the implementation phase were described.

This chapter concludes the description of the SLOOP method as it applies to the various phases of the software development life cycle. The next chapter deals with the incorporation of design patterns into a SLOOP design. The use of design patterns is not mandatory in a SLOOP design, but it can greatly enhance the reusability of the components of the system. Chapter 9 demonstrates the compatibility of the SLOOP approach with the concept of design patterns.

CHAPTER 9

INCORPORATING DESIGN PATTERNS INTO A SLOOP DESIGN

9.1 Introduction

As noted by Buschmann et al. [BMRSS96], one can classify patterns as being either **architectural patterns**, **design patterns** or **idioms**. Definitions of the various types of patterns were given in Chapter 3, Section 3.3.1.

The architectural patterns and design patterns listed in [BMRSS96] and [GHJV95] provide examples of many different types of design problems. The **purpose** of this chapter is to demonstrate that the **SLOOP approach** can be **applied successfully** to a **variety of design problems**. Several architectural and design patterns described in the above-mentioned references are therefore taken as examples and it is shown how they can be incorporated into a system that is designed using the SLOOP method.

Some of the patterns are already present in the original design of the call centre system as presented in Appendix B, while others can be added to improve the **reusability** and **extensibility** of the system. In both cases the SLOOP-specific issues are highlighted. These deal mainly with adherence to the SLOOP computational model, i.e. it has to be ensured that all the **parallel statements** of a particular application are **executed infinitely** often.

The purpose of incorporating architectural and design patterns into a SLOOP design is not to model the problem domain more accurately, but to yield a more reusable and flexible solution. The application of some of the design patterns results in reducing the amount of subclassing required when instantiating the system for a particular application.

It is beyond the scope of this chapter to provide detailed descriptions of the design patterns referenced here. For more information about these patterns the bibliographical references given in the various sections below should be consulted.

9.2 Architectural patterns

9.2.1 Pipes and filters

The design of the call centre system is reminiscent of the Pipes and Filters architectural pattern. The latter is described in [BMRSS96]. Briefly, data is generated by a data source, it is processed by successive filters and finally reaches a data sink. The filters are connected via pipes. Thus, the data is stored in a pipe by the previous filter or the source. The next filter in the pipeline then obtains the data from the pipe for further processing. This pattern is applicable when data is

processed in sequential steps. A filter processes its input incrementally, i.e. it does not read all its input before it processes the data and starts producing output. This promotes **parallelism** and a **low latency**.

In the call centre example shown in Figure 9-1, the CommsProviderSimulator[1] instance acts as the source of the data that needs to be processed (in this case a new service request is added to the `inputQ` (the first pipe in the pipeline). The ServiceCategoryAllocator[2] instance (the first filter) obtains the service request from the `inputQ` and initiates the categorisation of the service request. Once the service request has been categorised, it is assigned to the appropriate `serviceQ` (the next pipe). The next filter is the relevant ServiceCategory[3] instance, which assigns the service request to an idle ServiceProviderSimulator[4] instance. The latter acts as the data sink in this example.

One of the **benefits** of the Pipes and Filters pattern is that these **components** can be **added, deleted or rearranged** as required. In the call centre example the ServiceCategoryAllocator instance obtains its input from a FIFO queue of service requests. The ServiceCategoryAllocator instance receives the reference to its input pipe as a parameter when its `p_categorise:using:` method is invoked. It is therefore a trivial matter to construct an application where the ServiceCategoryAllocator obtains its input from a different queue. The only requirement is that the new queue should contain service requests of the same type.

It is also fairly simple to add another filter and a pipe between the ServiceCategoryAllocator and its existing input pipe. The new filter would process the service requests from the existing input pipe. Its output pipe would serve as the new input pipe of the ServiceCategoryAllocator.

Another possibility would be to construct a system where the ServiceCategoryAllocator was replaced by a completely different filter.

The SLOOP design approach is particularly suited to this architectural pattern, because it is based on a concept of a number of parallel statements that execute infinitely often. A parallel statement has an **effect** (e.g. it performs the filter function) if its ***if*-clause is true** (e.g. there is an element in the input queue). Parallel statements that do not share processors and that do not send messages to the same objects may execute **concurrently**. This facilitates the realization of the goals of increased parallelism and low latency.

[1] The CommsProviderSimulator class is specified in Appendix B, Section B.6.
[2] The ServiceCategoryAllocator class is specified in Appendix B, Section B.8.
[3] The ServiceCategory class is specified in Appendix B, Section B.10.
[4] The ServiceProviderSimulator class is specified in Appendix B, Section B.13.

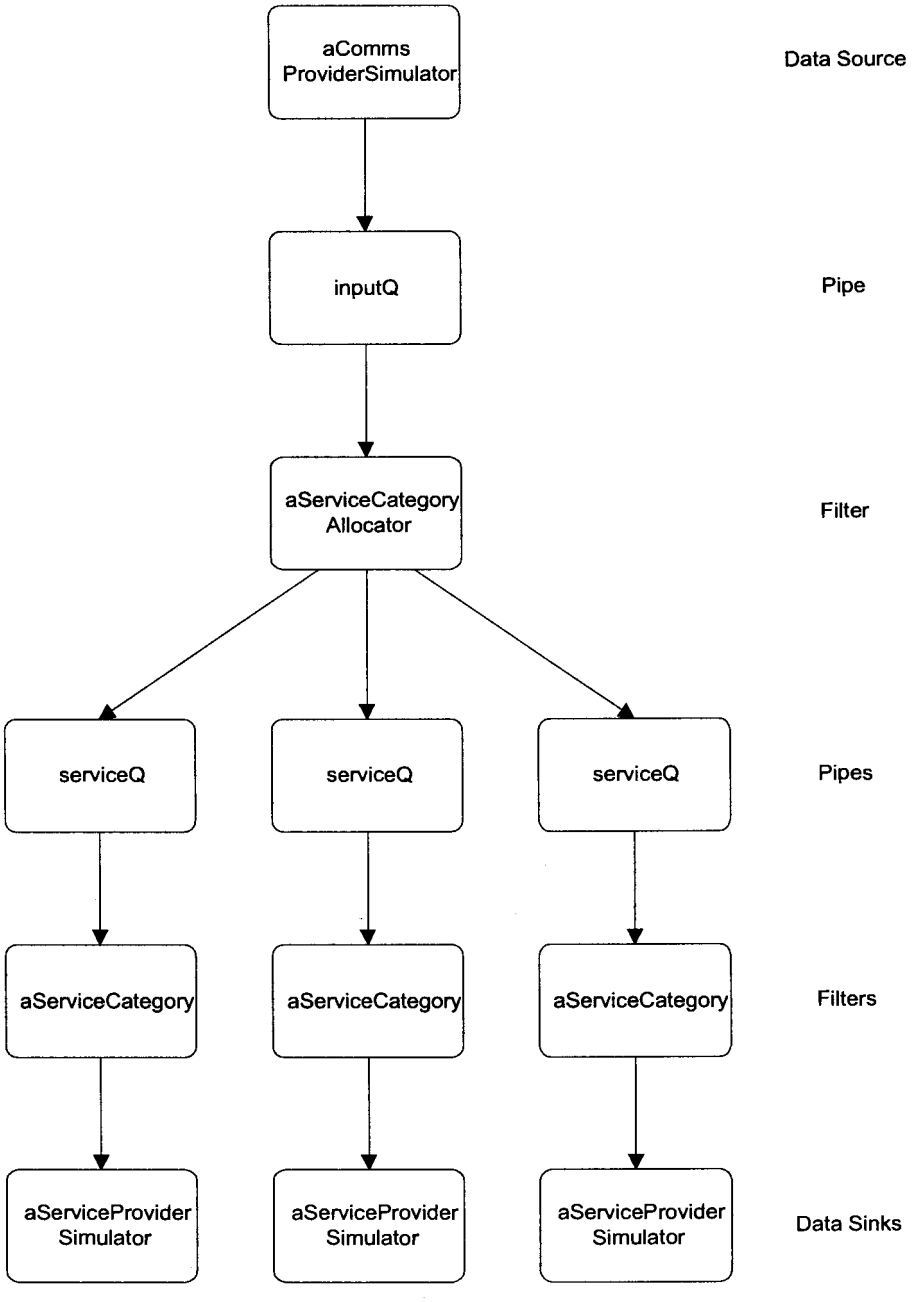

Figure 9-1. Pipes and filters in the call centre system.

9.2.2 Reflection

The reflection architectural pattern as described by [BMRSS96], provides a mechanism to change the behaviour of a software system by modifying the information at the meta-level. In CLOS, a reflective programming language [Keen89], the operations defined for an object are called generic functions. The invocation of such a generic function comprises a number of steps. First of all, the methods applicable to a given invocation of a generic function are determined, then they are sorted in decreasing order of precedence and subsequently a final sequence of methods is selected for execution [BMRSS96].

Similarly, in a SLOOP mapping which uses reflective facilities for its statement selection, the reflective computation **determines which of the private methods** associated with the parallel method **should be executed** at each parallel method invocation. (This was described in detail in Section 8.6.2 of the previous chapter.) A notable difference is that in a SLOOP mapping only one method may be selected for execution per parallel method invocation.

Some **assertion checking** can also be performed using reflective computation. Another possibility is the **generation of trace information**. These applications of reflective computation in the SLOOP method were discussed in Chapter 8, Section 8.6.3.

9.3 Creational design patterns

This section focuses on creational design patterns. The Factory Method and Singleton are two examples of creational design patterns [GHJV95]. The call centre example is used to demonstrate the **compatibility** between these patterns and the SLOOP approach towards system design.

9.3.1 The Factory Method

The CC_Activation class is used to instantiate the classes used by the call centre system and to ensure that it is done in the correct order. Evaluation of the design of the CC_Activation class as described in Chapter 6, Section 6.6.1, and in Appendix B, Section B.2, shows that it is already designed in the mould of two fundamental design patterns, viz. the Template Method (a behavioural design pattern) and the Factory Method (a creational design pattern). The aspects related to the Factory Method design pattern are described here. Section 9.5.3 covers the application of the Template Method design pattern.

The Factory Method design pattern is used when it is necessary for a class to **defer** the actual **instantiation** of certain classes to its subclasses. The **abstract class** has knowledge about the **sequence** in which it has to instantiate classes, but it needs to allow its **subclasses** to specify exactly **which** classes should be instantiated. Factory Methods are therefore used to instantiate those classes that are likely to be subclassed (or even replaced by other classes).

A Factory Method is usually invoked from within a **Template Method**. The latter indicates the **sequence** in which the classes should be instantiated. The Factory Methods are invoked at the various locations where these classes need to be instantiated. The **Factory Methods** therefore provide the **hooks for the instantiation** of these classes without committing to specific classes.

The `initialize` method of the CC_Activation class represents a Template method. It invokes several Factory Methods as can be seen below. Examples of objects that are created via Factory Methods are the `config` and `commsAgent` objects. In contrast the `userConnections` object is created directly by the CC_Activation class.

The statements of the `initialize` method of the CC_Activation class are as follows:

```
sequential
  config := self initManagement
[] commsAgent := self initCommsAgent
[] userConnections := SmalltalkLibPkg:::Array new: maxConn
[] < [] i where 1≤i≤maxConn :: userConnections at: i
       put: (self initConnection: i)
  >
[] inputQ := SmalltalkLibPkg:::OrderedCollection new: maxConn
[] scAllocator := self initServiceCategoryAllocator
[] scContainer := SmalltalkLibPkg:::Array new: maxCategories
```

```
[] < [] j where 1≤j≤maxCategories :: scContainer at: j
     put: (CC_CorePkg:::ServiceCategory setup: config)
   >
[] spAgentContainer := SmalltalkLibPkg:::Array new: maxSP
[] < [] k where 1≤k≤maxSP :: spAgentContainer at: k
     put: (self initSPAgent)
   >
[] timer:= SystemUtilitiesPkg:::TimerServices setup: config
[] timerEventQ := SmalltalkLibPkg:::OrderedCollection new
end-sequential
```

The `initCommsAgent` method of the CC_Activation class contains a single statement indicating that the subclass needs to reimplement the method. It is therefore an **abstract** method. The **correctness property** of this method indicates that a non-nil value will be returned, but it does **not** specify **which** class should be instantiated.

```
message pattern initCommsAgent
method properties
"Total correctness"
true results-in methodReturnValue notNil                    "DL1-05"
sequential
self subclassResponsibility
end-sequential
```

CC_SimulationActivation, a subclass of CC_Activation, **reimplements** this method. In this case the **correctness property** specifies **explicitly** which class is instantiated:

```
message pattern initCommsAgent
method properties
"Total correctness"
true results-in
      methodReturnValue notNil ∧
      CC_SimulationInterfacesPkg:::CommsProviderSimulator
      postconditions: (#startSimulation)
                                        "DL1-05 (CC_Activation)"
sequential
^CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation
end-sequential
```

The Factory method may either be **abstract** or it may contain a **default implementation**. The `initialize` method of the CC_Activation class contains examples of both. The `initCommsAgent` and `initSPAgent` methods are both abstract, whereas the other Factory Methods contain default implementations. For example, the initServiceCategoryAllocator Factory Method instantiates the ServiceCategoryAllocator class by default:

```
message pattern initServiceCategoryAllocator
method properties
"Total correctness"
true results-in
      methodReturnValue notNil                              "DL1-07"
sequential
^CC_CorePkg:::ServiceCategoryAllocator setup
end-sequential
```

Default methods are convenient when a reasonable default exists. When a new system has to be built, the methods do not have to be overridden in subclasses if the default implementation suffices. However, should the default implementation not be adequate, the design provides the **flexibility** to override only the relevant Factory Method(s). For example, instead of having to

override the `initialize` method of the CC_Activation class (and thereby creating the risk of modifying unrelated code unintentionally), only the `initServiceCategoryAllocator` method needs to be reimplemented if a subclass of the ServiceCategoryAllocator class needs to be instantiated.

Abstract methods are used where a default implementation is not feasible. For example, the class representing the communication provider functionality is likely to differ amongst the various applications. A communication provider simulator may even be used, as in the example in this thesis. For the same reason, the class representing the service provider functionality is instantiated via an abstract method.

A number of varieties of the Factory Method pattern are described in [GHJV95]. One of the disadvantages of the Factory Method is the fact that the class which invokes the Factory Method (in this case CC_Activation) has to be subclassed when any of the classes which it instantiates need to be subclassed.

A variation of the Factory Method which **avoids subclassing** defines instance variables to hold the class names of all the classes that need to be instantiated by the creating class. Each Factory Method now creates an instance of a class by referring to the contents of one of these variables. This alternative, as applied to the CC_Activation example, is shown below:

The following instance variables are defined in addition to the ones already specified for the CC_Activation class in Appendix B, Section B.2:

```
managementClass
cpAgentClass
connectionClass
scAllocatorClass
spAgentClass
```

The `initialize` method is modified by including the following statement as the first statement in that method:

```
self makeClasses
```

The `makeClasses` private method is implemented as follows:

```
message pattern makeClasses
method properties
"Total correctness"
true results-in methodReturnValue = self ∧
      managementClass notNil ∧
      cpAgentClass notNil ∧
      connectionClass notNil ∧
      scAllocatorClass notNil ∧
      spAgentClass notNil

sequential
answerString :=
      Dialog[5] request: 'Call centre configuration class: '

[]managementClass :=
      Class readFrom: (ReadStream on: answerString)

[]answerString :=
      Dialog request: 'Communication provider agent class: '
```

[5] The Dialog class is used to request typed input from the user [HoHo95].

```
[]cpAgentClass := Class readFrom: (ReadStream on: answerString)

[]answerString := Dialog request: 'Connection class: '
[]connectionClass :=
      Class readFrom: (ReadStream on: answerString)

[]answerString :=
      Dialog request: 'Service category allocator class: '
[]scAllocatorClass :=
      Class readFrom: (ReadStream on: answerString)

[]answerString :=
      Dialog request: 'Service provider agent class: '
[]spAgentClass := Class readFrom: (ReadStream on: answerString)
end-sequential
```

Thus, for each class a dialog box is displayed requesting the name of the class to be instantiated. The string that is provided by the user is then used to create an object that is the required class name. This name is stored in the appropriate instance variable. The Factory Methods that instantiate the various classes are all modified to use the values in the instance variables containing the class names. Two examples are given below; the first is of a method that previously was an abstract method and the second example is of a method that contained a default implementation.

```
message pattern initCommsAgent
method properties
"Total correctness"
cpAgentClass notNil results-in methodReturnValue notNil"DL1-05"
sequential
^cpAgentClass setup
end-sequential

message pattern initServiceCategoryAllocator
method properties
"Total correctness"
scAllocatorClass notNil results-in
      methodReturnValue notNil                          "DL1-07"
sequential
^scAllocatorClass setup
end-sequential
```

Note that the precondition of each method now requires that the relevant instance variable containing the class name should contain a non-nil value. The **correctness properties** of the method still **do not refer explicitly** to the specific class that is instantiated. There is now **no need to subclass** the CC_Activation class in order to instantiate the appropriate communication provider and service category allocator classes.

If a design is used which employs instance variables to store the class names, then it would be prudent to use generic names such as `setup` for all the instance creation methods. A name such as `startSimulation` would therefore not be a good choice, since it implies that the object will be performing a simulation. However, if a class with such an instance creation method name already exists (as in the case of the CommsProviderSimulator class), one can always add a method called `setup` to the set of methods supported by the class. The `setup` method can then invoke the `startSimulation` method.

There are a number of reasons why the CC_Activation class might have to be subclassed. The application of the Factory method as described above enables one to eliminate the need for

subclassing when the CC_Activation class does not know in advance which **classes** will have to be **instantiated**.

Another reason why the CC_Activation class might have to be subclassed is the fact that it also does not know in advance which **parallel methods** need to be **activated**. The classes that are instantiated determine the parallel methods that have to be activated. If the client **interface** of the parallel methods remains the same for a class and its subclasses, the need for subclassing in order to activate the appropriate parallel methods is also eliminated. This aspect will be discussed in more detail in Section 9.5.3.

9.3.2 Singleton

The Singleton design pattern [GHJV95] is useful when it is necessary to restrict the number of instances of a class to one. The pattern allows this sole instance to be referenced globally without requiring the use of a global variable.

One possible implementation of the pattern is the following: The class that is designed as a Singleton has a class variable that contains the reference to the sole instance of that class. Whenever the instance has to be obtained, the instance creation method is invoked. The latter checks whether the class variable contains the value nil. If it does, a new instance is created and returned, otherwise the existing instance is returned.

When incorporating this design pattern into a SLOOP design an interesting question arises. Since the class that is implemented as a Singleton does **not necessarily** have to be **instantiated during activation**, a way has to be found to ensure that its **parallel statements are activated** if there are any associated with the class. This issue will be addressed shortly. First an example of a class designed as a Singleton is presented.

The Singleton design pattern is often used in conjunction with the Flyweight and State patterns [GHJV95]. These are described in detail in Sections 9.4.2 and 9.5.2 respectively. The State design pattern is used to extract state-specific behaviour from a class. A separate class is defined for each state of the original class. The latter then becomes the context of the state. The Flyweight pattern is used to make the instances of the state classes shareable by multiple instances of the context class. Only one instance of each state class is therefore required. The application of the Singleton design pattern ensures that only one instance is created for each state class.

When the State pattern is applied to the Connection[6] class of the call centre system, the IdleConnection, ConnectedConnection and TerminatingConnection classes emerge as state classes (the rationale for defining these particular classes is given in Section 9.5.2). The Connection class then becomes their context class. The application of the Flyweight pattern ensures that the state classes can be shared by multiple instances of the Connection class. Each state class can then be implemented as a Singleton.

The instance creation methods of the IdleConnection class demonstrate how the Singleton design pattern ensures that only one instance of the class is instantiated. The **new** method that is usually used for instance creation is overridden with the **instance** method as shown below:

```
class IdleConnection
superclass ConnectionState
class variable names
      IdleConnectionInstance
      "This variable is used to implement the class as a Singleton"
```

[6] The SLOOP specification of the Connection class is presented in Appendix B, Section B.7.

```
class properties
      invariant   IdleConnection instanceCount ≤ 1
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."

class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in methodReturnValue = IdleConnectionInstance ∧
            IdleConnectionInstance notNil
      sequential
      IdleConnectionInstance := super new
            if IdleConnectionInstance isNil
      [] ^IdleConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in IdleConnection postconditions: (#instance)
      sequential
      ^ IdleConnection instance
      end-sequential
```

When an invariant describes a property of an **instance** of a class, the invariant only needs to hold once the instance has been created and initialized. However, in the above example, the invariant describes a property of the **class**. This invariant has to hold at all times.

Instead of using a global variable to refer to the IdleConnection instance, all clients access the object via the message expression `IdleConnection instance`. This obviates the need to instantiate the IdleConnection class during initialization. The instance is created when it is used the first time.

As described above, state classes are often implemented as Singletons. Since many of these states are only reached after initialization has completed, the corresponding classes are only instantiated at that time. For example, the TerminatingConnection class, which is implemented as a Singleton, represents the behaviour of a connection when its state has changed to 'TERMINATING'. This class is only instantiated when the 'TERMINATING' state is entered for the first time.

Since the class is not necessarily instantiated during activation, it raises the issue of the **activation** of the parallel statements associated with the class, if there are any. The designer has to ensure that the requirements for SLOOP parallel statements are met at all times, i.e. all parallel statements required by the program have to be executed infinitely often. In order to facilitate reasoning about correctness, these statements have to be available for scheduling at all times after the instantiation of the class.

The only exception is when they are used in a very specific way, as described in Chapter 4, Section 4.3.5.3. To recapitulate: if a conditional expression is associated with a parallel statement, the statement has to be available for scheduling whenever the conditional expression evaluates to true. When the conditional expression evaluates to false, the statement need not be present, since the execution of the statement will have no effect even if it is present.

In the case of the Connection class, its parallel statements only have an effect if the Connection instance is in the 'TERMINATING' state, as can be seen from the cyclic methods listed in Appendix B, Section B.7. Thus, when these statements are moved to the TerminatingConnection class, it only needs to be guaranteed that these statements are available for scheduling **whenever** the Connection instance is in the 'TERMINATING' state, i.e. whenever the `state` instance variable of the Connection instance refers to the TerminatingConnection instance.

The statements in the `p_executeConnection:` method of the CC_Activation class are responsible for activating the parallel methods of the TerminatingConnection class, viz. the `p_informCommsProvider:context:` and `p_doWrapUp:` methods. In the statements below, these methods are only invoked if the connection is in the 'TERMINATING' state.

```
aConnection state p_informCommsProvider: commsAgent
context: aConnection
     if aConnection state = TerminatingConnection someInstance[7]
[] aConnection state p_doWrapUp: aConnection
     if aConnection state = TerminatingConnection someInstance
```

The parallel methods of the TerminatingConnection class are therefore invoked via the parallel statements in the *activation-section* of the program[8], but their invocation is subject to an instance of the TerminatingConnection class currently being in use by the relevant Connection instance. The **effect** that is achieved is therefore similar to that which applied when the statements were still part of the Connection class.

Thus, when the software designer incorporates a Singleton design pattern into a SLOOP design, it has to be done in such a way that the parallel statements that are involved can still be scheduled whenever they can have an effect. The way in which this can be achieved was demonstrated above.

Further clarification of the above example will be given in Section 9.5.2, where the State design pattern is explained in more detail.

9.4 Structural design patterns

9.4.1 Adapter

The communication provider and the service providers form part of the environment of the call centre system. The interfaces of these objects are therefore not determined by the designer of the call centre. These interfaces may vary, depending on the product being used. However, it is desirable to present a consistent interface to the clients of these objects within the call centre system. For this reason the CommsProviderAgent and ServiceProviderAgent classes are introduced. The clients within the call centre system deal with the agents only. The agents adapt the messages received from the clients to suit the interfaces of the actual communication and service providers being used. The agent classes are subclassed as needed, based on the interfaces of the communication and service provider classes. This is an example of an application of the Adapter design pattern. The latter is described in detail in [GHJV95].

In the SLOOP program given in Appendix B, the functionality of the agents is simulated via the CommsProviderSimulator and ServiceProviderSimulator classes. The interfaces presented to the communication and service provider clients are the same as those that should be presented by the corresponding CommsProviderAgent and ServiceProviderAgent classes. The latter may

[7] The someInstance method of the Smalltalk Behavior class returns an existing instance of the receiver.
[8] The p_activate message is sent to the CC_SimulationActivation instance from within the *activation-section* of the program. In turn, the p_activate method invokes the p_executeConnection: method inherited from the CC_Activation class.

therefore be defined as descendants of the respective simulation classes. Only the methods that perform the simulations need to be overridden.

There are no special considerations involved when applying this pattern to a SLOOP design.

9.4.2 Flyweight

One of the situations in which the Flyweight design pattern is applicable is when there is a proliferation of objects that may be reduced if some of these objects can be made shareable [GHJV95]. In order to use an object in multiple contexts simultaneously (i.e. as a Flyweight), it is necessary to store all information that is dependent on the context of the shared object in the context itself (this is called the extrinsic state of the Flyweight).

The Flyweight pattern is often used in conjunction with the State pattern. The latter implements the various states defined for a class as separate classes. The original class becomes the context of these state classes. All actions that are dependent on the state of the context are performed by the appropriate state objects. There is an instance of each state class for each instance of the context class. If the number of context instances is high and there are numerous states, then the application of the State design pattern can result in an unacceptably high number of objects. If the context-specific data can be stored in the context, then the instances of the state classes can be shared by multiple contexts, i.e. each state class can be implemented as a Flyweight.

This design pattern can be used successfully in a SLOOP design, provided that each parallel statement associated with the Flyweight is still executed infinitely often for each instance of the context class once the SLOOP program is **mapped** to an executable program. The way in which this can be achieved is discussed next.

The mapping of a SLOOP program to various architectures was discussed in Chapter 8. Each invocation of a parallel method results in the execution of one of its statements. The mapping algorithm has to guarantee that each statement of each activated parallel method will be executed infinitely often. One way of achieving this is by implementing an instance variable that keeps track of the last statement that was executed.

Since it has to be guaranteed that **each parallel statement** is executed infinitely often **for each context**, this variable (called the **statement selector) has to form part of the context**. The following scenario illustrates the problem that would occur if it were present in the state class instance (i.e. shared by multiple contexts).

Suppose there are four Connection instances and they are all in the same state, i.e. there are four context objects sharing a single instance of a state class. The latter has a single parallel method containing four statements. Suppose further that the parallel method is invoked for each Connection instance in a round robin fashion. This implies that during the first rotation the statements will be allocated to the various contexts as follows if the statement selector is shared by the contexts:

Context 1: statement 1
Context 2: statement 2
Context 3: statement 3
Context 4: statement 4

Unfortunately, the subsequent rotations will follow the same pattern, which means that for any given context, three of the statements will never be executed. It is therefore clear that the statement selector is context-specific and should be implemented as such. An example of the application of the Flyweight pattern in the call centre system is given in Section 9.5.2, where the State pattern is discussed.

9.4.3 Proxy

One of the applications of the Proxy design pattern is to allow one object to act as a local representative of a remote object [GHJV95]. It does not adapt one interface to another. This pattern is used extensively in distributed system infrastructures such as CORBA [BMRSS96]. Objects send messages to other objects without taking the location of the target objects into account, i.e. whether they are local or remote is irrelevant to the client object.

One possible implementation of such a system requires the instantiation of a proxy per address space for each object. The syntax of the messages sent to the proxy is exactly the same as for those sent to the original object. The proxy is responsible for obtaining information about the physical location of the original object and for performing the functions related to converting the message into a format that can be transmitted to a remote processor. These aspects are transparent to the client of the remote object.

This architecture is reused when SLOOP programs are **mapped** to distributed systems. As described in Chapter 8, SLOOP programs are designed in a unified manner. The physical location of each object is irrelevant. During the implementation phase the target architecture is determined. At that stage issues such as how to ensure the atomicity of SLOOP statements are addressed. Once a parallel statement has been selected for execution and all the required resources have been reserved for it, design patterns such as Proxy make it possible to send messages to local and remote objects within these statements without having to take their locations into account. Thus, the SLOOP program can be mapped to a distributed system without having to modify the design of the SLOOP program.

9.5 Behavioural design patterns

9.5.1 Iterator

The Iterator design pattern facilitates sequential access to the elements of a collection while hiding the underlying representation from the client [GHJV95]. Most class libraries provide this functionality for their collection classes. The Smalltalk internal Iterator method[9] `do:` is one example. This method evaluates the block that is supplied as argument to this method for each element of the collection representing the receiver of this message. The Iterator design pattern is therefore automatically used when SLOOP programs contain Smalltalk message expressions that invoke this method.

9.5.2 State

The Connection class of the call centre system maintains a state instance variable. If an event occurs, the behaviour depends on the current state of the object. The disadvantage of this design is the fact that the addition of a new state implies that each method that selects behaviour based on the current state of the object needs to be modified to include the behaviour for the new state.

The State design pattern [GHJV95] offers an alternative solution: the state-specific behaviour is removed from the original class. The latter becomes the **context** of a new class hierarchy. An abstract superclass is defined which represents the aspects that are generic to the states of the original class. Each state of the original class is implemented as a subclass of the new abstract

[9] Gamma et al. [GHJV95] define an internal iterator as one where the iterator controls the iteration. The client is responsible for advancing the traversal in the case of an external iterator.

class, therefore each method handling an event only contains the logic for the state represented by that particular class. The addition of a new state simply means the addition of a new subclass.

The following example shows how the **extensibility** of the Connection class is improved by using the State design pattern.

A new **abstract** class ConnectionState is defined, as shown in Figure 9-2. The Connection class becomes the **context** of the ConnectionState class. Three **concrete subclasses** are defined for the ConnectionState abstract class, viz. IdleConnection, ConnectedConnection and TerminatingConnection. They encapsulate the behaviour corresponding to the 'IDLE', 'CONNECTED' and 'TERMINATING' states previously defined within the Connection class. These classes are implemented as Flyweights, i.e. all the context-specific information is stored within the Connection class. (The application of the Flyweight design pattern was described in Section 9.4.2.)

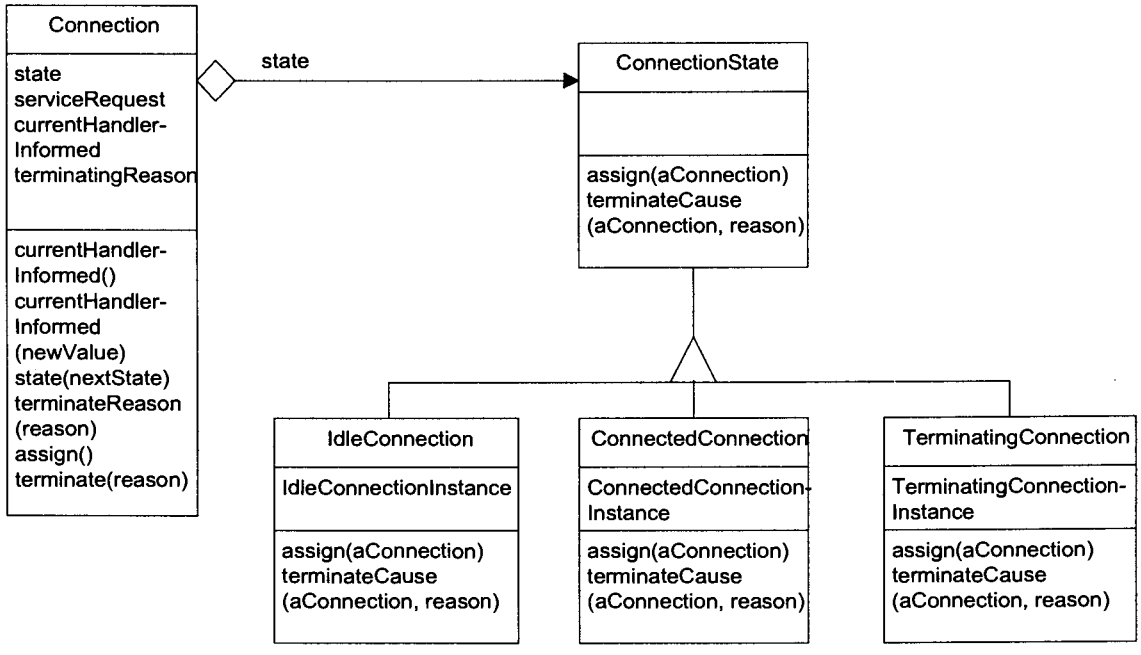

Figure 9-2. Incorporating the State design pattern into the Connection class.

The Connection instance still maintains an instance variable called state. However, instead of containing the values 'IDLE', 'CONNECTED' or 'TERMINATING', it now contains a reference to an instance of IdleConnection, ConnectedConnection or TerminatingConnection. When the Connection instance has to execute state-specific logic, it simply invokes the relevant method of the ConnectionState subclass instance that is currently referenced by its state instance variable.

The SLOOP specification of the original Connection class is given in Appendix B, Section B.7. The modifications to the methods of the Connection class are now presented (the modified parts are shown in bold italics):

```
category private
      message pattern initialize: indexOfConnection
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧
      state = IdleConnection instance ∧
      serviceRequest notNil ∧ currentHandlerInformed = false ∧
      connectionIndex = indexOfConnection                          "DL1-11"
      sequential
      state := IdleConnection instance
      [] serviceRequest := CC_CorePkg:::ServiceRequest setup: self
      [] currentHandlerInformed := false
      [] connectionIndex := indexOfConnection
      end-sequential
```

When the Connection class is initialized, the state variable now contains a reference to an instance of the IdleConnection class. The method `instance` used here is a class method of the IdleConnection class. Since the latter is implemented as a Singleton (as described in Section 9.3.2), all accesses to the IdleConnection instance are via this method.

The next two methods are used by the clients of the Connection instance to determine whether the latter is idle or busy terminating. The Smalltalk `someInstance` method returns an existing instance of the receiver. Due to the application of the Singleton design pattern it is guaranteed that there will never be more than one instance of each of the ConnectionState subclasses. The statements in the methods below therefore suffice to provide the required answers.

```
category testing
      message pattern isIdle
      method properties
      "Total correctness"
      true results-in methodReturnValue =
            (state = IdleConnection someInstance)                  "DL1-05"
      sequential
      ^ state = IdleConnection someInstance
      end-sequential

      message pattern isTerminating
      method properties
      "Total correctness"
      true results-in methodReturnValue =
            (state = TerminatingConnection someInstance)           "DL1-06"
      sequential
      ^ state = TerminatingConnection someInstance
      end-sequential
```

Previously, accessing methods were provided to obtain the values of some of the instance variables of the Connection class, viz. `terminatingReason`, `serviceRequest` and `connectionIndex`. The values of the `state` and `currentHandlerInformed` instance variables were only significant to the Connection instance itself, therefore no accessing and modifying methods were provided for these variables. The introduction of the ConnectionState subclasses results in new accessing and modification methods being required, since actions that previously had been performed within the Connection class are now initiated externally (i.e. from within the ConnectionState subclasses). The new methods are shown next.

```
category accessing
      message pattern currentHandlerInformed
      method properties
      "Total correctness"
      true results-in methodReturnValue = currentHandlerInformed
```

```
        sequential
        ^currentHandlerInformed
        end-sequential

category modifying
        message pattern terminateReason: reason
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧
                terminateReason = reason
        sequential
        terminateReason := reason
        end-sequential

        message pattern state: nextState
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧ state = nextState
        sequential
        state := nextState
        end-sequential

        message pattern currentHandlerInformed: newValue
        method properties
        "Total correctness"
        true results-in
        methodReturnValue = self ∧ currentHandlerInformed = newValue
        sequential
        currentHandlerInformed := newValue
        end-sequential
```

The methods that contain state-specific logic are the `assign` and `terminate:` methods in the modifying category, as well as the `p_informCommsProvider:` and `p_doWrapUp` methods in the cyclic category. The functionality of these methods is now delegated to the ConnectionState subclasses, as evident from the modified methods below:

```
category modifying
        message pattern assign
        method properties
        "Total correctness"
        state = IdleConnection someInstance results-in
                methodReturnValue = self ∧
                state postconditions: (#assign:) withArguments: #(self)
                                                                "DL1-07"
        sequential
        state assign: self
                if state = IdleConnection someInstance
        end-sequential

        message pattern terminate: reason
        method properties
        "Total correctness"
        state = ConnectedConnection someInstance results-in
                methodReturnValue = self ∧
                state postconditions: (#terminate:cause:)
                withArguments: #(self reason)                   "DL1-08"
        "Total correctness"
        state = TerminatingConnection someInstance results-in
                methodReturnValue = self                        "DL1-09"
        "This allows for terminate collision."
```

```
"Total correctness"
state = IdleConnection someInstance results-in
      methodReturnValue = self                          "DL1-10"
"This ensures that the transition from 'IDLE' to 'TERMINATING'
is not possible"
sequential
state terminate: self cause: reason
      if state = ConnectedConnection someInstance
end-sequential
```

The parallel methods are removed from the Connection class, since they are dependent on the state of the Connection instance.

The new ConnectionState class and its subclasses are now shown (the only class containing parallel methods is the TerminatingConnection class):

```
"---------------------------------------- class ConnectionState ---"
class ConnectionState
superclass Object
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      "The set of ConnectionState subclasses and the allowed state
      transitions are not specified here, because it would require
      ConnectionState to be subclassed if this was changed."
instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = IdleConnection someInstance results-in
            methodReturnValue = self ∧
            aConnection state postconditions: (#assign:)
            withArguments: #(aConnection)                "DL1-01"
      sequential
      self subclassResponsibility
      end-sequential

      message pattern terminate: aConnection cause: reason
      method properties
      aConnection state notNil results-in
            methodReturnValue = self ∧
            aConnection state postconditions: (#terminate:cause:)
            withArguments: #(aConnection reason)          "DL1-02"
      sequential
      self subclassResponsibility
      end-sequential
```

```
"---------------------------------------- class IdleConnection ---"
class IdleConnection
superclass ConnectionState
class variable names
      IdleConnectionInstance
      "This variable is used to implement the class as a Singleton"
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      invariant    IdleConnection instanceCount ≤ 1          "DS2-01"
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."
class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = IdleConnectionInstance ∧
            IdleConnectionInstance notNil                    "DL1-01"
      sequential
       IdleConnectionInstance := super new
            if IdleConnectionInstance isNil
      [] ^IdleConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in IdleConnection postconditions: (#instance)
                                                             "DL1-02"
      sequential
      ^ IdleConnection instance
      end-sequential

instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = IdleConnection someInstance results-in
            methodReturnValue = self ∧
            aConnection postconditions: (#state:)
            withArguments #((ConnectedConnection instance))
                                          "DL1-01(ConnectionState)"
      sequential
      aConnection state: (ConnectedConnection instance)
      end-sequential

      message pattern terminate: aConnection cause: reason
      method properties
      "Total correctness"
      aConnection state = IdleConnection someInstance results-in
            methodReturnValue = self       "DL1-02(ConnectionState)"
      "This ensures that the transition from 'IDLE' to 'TERMINATING'
      is not allowed"
      sequential
      ^self
      end-sequential
```

```
"------------------------------------ class ConnectedConnection ---"
class ConnectedConnection
superclass ConnectionState
class variable names
      ConnectedConnectionInstance
      "This is used to implement the class as a Singleton"
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      invariant   ConnectedConnection instanceCount ≤ 1      "DS2-01"
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."

class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = ConnectedConnectionInstance ∧
            ConnectedConnectionInstance notNil                "DL1-01"
      sequential
       ConnectedConnectionInstance := super new
            if ConnectedConnectionInstance isNil
      [] ^ConnectedConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in
      ConnectedConnection postconditions: (#instance)         "DL1-02"
      sequential
      ^ ConnectedConnection instance
      end-sequential

instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = ConnectedConnection someInstance results-in
            methodReturnValue = self       "DL1-01 (ConnectionState)"
      "This ensures that the transition from 'CONNECTED' to
      'CONNECTED' is not allowed"
      sequential
      ^self
      end-sequential

      message pattern terminate: aConnection cause: reason
      method properties
      "Total correctness"
      aConnection state = ConnectedConnection someInstance results-in
            methodReturnValue = self ∧
            aConnection postconditions: (#state:)
            withArguments: #((TerminatingConnection instance)) ∧
```

```
            aConnection postconditions: (#terminateReason:)
            withArguments: #(reason)        "DL1-02 (ConnectionState)"
      sequential
      aConnection terminateReason: reason
      [] aConnection state: (TerminatingConnection instance)
      end-sequential


"---------------------------------- class TerminatingConnection ---"
class TerminatingConnection
superclass ConnectionState
class variable names
      TerminatingConnectionInstance
      "This is used to implement the class as a Singleton"
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      invariant   TerminatingConnection instanceCount ≤ 1     "DS2-01"
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."

class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = TerminatingConnectionInstance ∧
            TerminatingConnectionInstance notNil              "DL1-01"
      sequential
       TerminatingConnectionInstance := super new
            if TerminatingConnectionInstance isNil
      [] ^TerminatingConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in
      TerminatingConnection postconditions: (#instance)       "DL1-02"
      sequential
      ^ TerminatingConnection instance
      end-sequential

instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = TerminatingConnection someInstance
            results-in methodReturnValue = self
                                          "DL1-01(ConnectionState)"
      "This ensures that the transition from 'TERMINATING' to
      'CONNECTED' is not allowed"
      sequential
      ^self
      end-sequential
```

```
        message pattern terminate: aConnection cause: reason
        method properties
        "This is the terminate collision case.  It is important not to
        overwrite terminateReason with reason, since the connection is
        already busy terminating."
        "Total correctness"
        aConnection state = TerminatingConnection someInstance
                results-in methodReturnValue = self
                                                "DL1-02(ConnectionState)"
        "This takes care of terminate collision."
        sequential
        ^self
        end-sequential

category cyclic
        message pattern p_informCommsProvider: commsAgent
                context: aConnection[10]
        method properties
        "Safe liveness"
        aConnection state = TerminatingConnection someInstance ∧
        aConnection terminatingReason = 'completed' ∧
        ¬(aConnection currentHandlerInformed) ensures
                commsAgent postconditions: (#terminate:cause:)
                withArguments:
                #(aConnection (aConnection terminatingReason)) ∧
                aConnection currentHandlerInformed              "DP1-01"
        parallel
        commsAgent terminate: aConnection
        cause: (aConnection terminatingReason) \+
        aConnection currentHandlerInformed: true
                if aConnection terminatingReason = 'completed'
                and: [aConnection currentHandlerInformed not]
        end-parallel

        message pattern p_doWrapUp: aConnection
        method properties
        "Safe liveness"
        aConnection currentHandlerInformed ensures
                aConnection state = IdleConnection someInstance ∧
                (aConnection serviceRequest) postconditions: (#reset) ∧
                ¬(aConnection currentHandlerInformed)           "DP1-02"
        parallel
        aConnection serviceRequest reset \+
        aConnection state: IdleConnection someInstance \+
        aConnection currentHandlerInformed: false
                if aConnection currentHandlerInformed
        end-parallel
```

The above implementation of the State design pattern provides an example of the usage of dynamic parallel statements.

In the original SLOOP program given in Appendix B, the Connection class contained two parallel methods, viz. `p_informCommsProvider:` and `p_doWrapUp`. These methods were invoked infinitely often for each Connection instance due to the presence of the following statements in the `p_activate` method of the CC_Activation class:

[10] The p_informCommsProvider: and p_doWrapUp methods are modified to include an additional argument, viz. the context.

```
[] < [] i where 1≤i≤maxConn ::
        self p_executeConnection: (userConnections at: i)
  >
```

The original p_executeConnection: method of the CC_Activation class contained the following statements:

```
"statements of the original p_executeConnection: method"
parallel
aConnection p_informCommsProvider: commsAgent
[] aConnection p_doWrapUp
end-parallel
```

The parallel statements of the Connection class only have an effect if the connection is in the 'TERMINATING' state, otherwise none of the **assignments** or **modifying** message expressions are executed (only the *if* clauses are executed). The parallel methods of the original Connection class are repeated here for easy reference. Those parts of the *if* clauses that refer to the state of the connection are highlighted in bold italics.

```
message pattern p_informCommsProvider: commsAgent
method properties
"Safe liveness"
state = 'TERMINATING' ∧ terminatingReason = 'completed' ∧
¬currentHandlerInformed ensures
        commsAgent postconditions: (#terminate:cause:)
        withArguments: #(self terminatingReason) ∧
        currentHandlerInformed                          "DP1-01"
parallel
commsAgent terminate: self cause: terminatingReason \+
currentHandlerInformed := true
        if state = 'TERMINATING' and:
        [(terminatingReason = 'completed')
        and: [currentHandlerInformed not]]
end-parallel

message pattern p_doWrapUp
method properties
"Safe liveness"
currentHandlerInformed ensures
state = 'IDLE' ∧ serviceRequest postconditions: (#reset) ∧
¬currentHandlerInformed                                 "DP1-02"
parallel
state := 'IDLE' \+
serviceRequest reset \+
currentHandlerInformed := false
        if currentHandlerInformed
"By following the logic of the methods of the Connection class
it will be evident that currentHandlerInformed can only be true
while the connection is in the 'TERMINATING' state"
end-parallel
```

When the State design pattern is implemented, the p_informCommsProvider: and p_doWrapUp methods are moved to the TerminatingConnection class. They are not present in the other State subclasses. Thus, when the connection is not in the 'TERMINATING' state (i.e. its state variable refers to a ConnectionState subclass instance other than the TerminatingConnection instance), the statements of these methods are not part of the list of parallel statements that are executed infinitely often.

The statements in the `p_executeConnection:` method of the CC_Activation class are changed to:

```
(aConnection state) p_informCommsProvider: commsAgent
context: aConnection
      if aConnection state = TerminatingConnection someInstance
[] aConnection state p_doWrapUp: aConnection
      if aConnection state = TerminatingConnection someInstance
```

Thus, the statements of the `p_informCommsProvider:context:` and `p_doWrapUp:` methods are present in the list of parallel statements whenever the Connection instance contains a reference to the TerminatingConnection instance (i.e. it is in the 'TERMINATING' state).

The statements of the `p_informCommsProvider:context:` and `p_doWrapUp:` methods no longer have to check the state of the Connection instance, since they can only be invoked if the TerminatingConnection instance is active.

This concludes the discussion of the State design pattern. The issues related to the SLOOP computational model were highlighted and it was shown why they did not present a problem. The above example has therefore demonstrated that the State design pattern can be applied successfully to a design based on the SLOOP method.

9.5.3 Template

The Template Method is used to implement the **skeleton of an algorithm**, allowing some steps to be reimplemented by subclasses [GHJV95]. This is achieved by invoking **abstract** methods or **default** methods for some steps of the algorithm. That way it ensures that all the necessary steps are executed and that they are performed in the correct order, while allowing subclasses to redefine the variant parts of the algorithm.

In Section 9.3.1 it was mentioned that the `initialize` method of the CC_Activation[11] class was implemented as a Template Method. In that section the focus was on the Factory Methods that were used as abstract or default methods. The discussion here concentrates on the Template Method characteristics of the design of the `initialize` method.

The method includes all the statements that are necessary to instantiate all the classes required by the system. The order in which these statements appear ensures that the correctness properties specified for the method are satisfied. For example, the `config` object (created via the `initManagement` method) has to exist prior to the instantiation of many of the other classes, since configuration information is obtained from the `config` object during the instantiation of these classes. By reusing the `initialize` method, the designer is assured of performing the instantiation actions in the correct order.

The *activation-section* of a SLOOP program not only contains statements to instantiate the relevant classes, but it also **activates** the appropriate **parallel statements**. These parallel statements may vary, depending on the classes that are instantiated. The Template Method design pattern is therefore also used in the `p_activate` method of the CC_Activation example. It contains the following statements:

[11] The CC_Activation class is specified in Appendix B, Section B.2.

```
parallel
self p_executeCPAgent
"Execute the parallel statements of the commsAgent."

[] timer p_runTimer: timerEventQ
"Whenever a timeout occurs, the TimeoutElement instance
representing the timeout is added to the end of the timerEventQ,
which indicates to the requestor that the specified timer has
expired."

[] self p_categoriseAndAllocate
"Once a service request has been categorised, it is removed from
the inputQ and appended to the appropriate serviceQ."

[] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
      p_execute
  >
"For each service category the associated service queue and set
of service provider agents are monitored.  If the service queue
is not empty and a service provider agent in the spSubset
associated with the service category is available to process a
new service request, the first element of the service queue is
removed and assigned to a service provider agent."

[] < [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
  >
"When a connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection.  Once all the procedures have been completed to
terminate the connection, the connection and its associated
service request are reset to their initial states."

[] < [] k where 1≤k≤maxSP :: self p_executeSPAgent:
      (spAgentContainer at: k)
  >
"Execute the parallel statements of the service provider
agents."

end-parallel
```

As mentioned earlier, the Template Method may invoke abstract or default methods. The example above illustrates both types of invocations. The `p_executeCPAgent` method is abstract, whereas the `p_categoriseAndAllocate` method is a default method. The implementation of each method is shown below:

```
message pattern p_executeCPAgent
method properties
"These are the properties pertaining to the communication
provider interface as identified during the analysis phase."
parallel
self subclassResponsibility
end-parallel

message pattern p_categoriseAndAllocate
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
```

```
parallel
scAllocator p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] scAllocator p_allocate: scContainer from: inputQ
"Once the service request has been categorised, the scAllocator
removes it from the inputQ and appends it to the appropriate
serviceQ."
end-parallel
```

By using the Template Method design pattern, it is clear to the designer of a new application that the parallel statements of the `cpAgent` object should be included, but the actual statements are only specified once the relevant interface class is determined. In the case of the `p_categoriseAndAllocate` method a default implementation is feasible, which is provided for the convenience of the designers of future applications.

As described in Section 9.3.1, subclassing can be avoided when using certain variants of the Factory method during instance creation. This is only beneficial if the parallel methods belonging to the classes being instantiated can also be activated without having to resort to subclassing. This implies that although the set of parallel methods may differ for each class, they have to be activated via the same message expression. One way of achieving this is by **encapsulating** the parallel methods of each subclass in such a way that **all subclasses present the same interface** to the client.

For example, a subclass of the ServiceCategoryAllocator[12] class might obtain information from a database in order to categorise a service request. In order to accomplish this, it may be necessary to define additional parallel methods for the ServiceCategoryAllocator subclass as well as modify the ones inherited from the parent class. These changes can be hidden from the client if the parallel methods of the ServiceCategoryAllocator class and its subclasses are encapsulated in a new method, viz. `p_categorise:allocate:`

The `p_categoriseAndAllocate` method of the CC_Activation class now only refers to this encapsulating method, as shown below:

```
message pattern p_categoriseAndAllocate
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
parallel
scAllocator p_categorise: inputQ allocate: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request).  Once it has been categorised, it removes it from the
inputQ and appends it to the appropriate serviceQ."
end-parallel
```

The `p_categorise:allocate:` method of the **ServiceCategoryAllocator** class now invokes the methods previously invoked by the `p_categoriseAndAllocate` method of the **CC_Activation** class:

```
message pattern p_categorise: inputQ
      allocate: scContainer
```

[12] The ServiceCategoryAllocator class is specified in Appendix B, Section B.8.

```
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
parallel
self p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] self p_allocate: scContainer from: inputQ
"Once the service request has been categorised, the scAllocator
removes it from the inputQ and appends it to the appropriate
serviceQ."
end-parallel
```

Subclasses of the ServiceCategoryAllocator class may alter the implementation of the `p_categorise:allocate:` method (e.g. by adding parallel statements to obtain information from a database) without affecting the CC_Activation class.

Note that the encapsulating method has to pass all the arguments required by the encapsulated parallel statements. This interface remains the same, even though some subclasses may not require all the arguments.

9.5.4 Strategy

Before a service request can be allocated to one of the call centre service queues, its category has to be determined. Different applications may require different categorisation algorithms to be used. For example, the information could be extracted from the service request itself, a database could be consulted or the call centre could enter into a dialogue with the service user to obtain the information. The Strategy pattern is useful to allow one to vary the algorithm without requiring subclassing.

The `categoriseServiceRequest:using:` method of the ServiceCategoryAllocator class implements the categorisation algorithm in the call centre system. In the original SLOOP program given in Appendix B, the ServiceCategoryAllocator class needs to be subclassed if the default implementation of this method presented in Section B.8 does not suffice.

Incorporating the Strategy pattern involves the definition of a new class, viz. CategorisingStrategy. It represents the common interface used by the context when the latter invokes one of the supported algorithms. The subclasses of this class represent the various options as listed above, as well as any future implementations. The abstract class is defined as follows:

```
class CategorisingStrategy
superclass Object
instance methods
category modifying
      message pattern categoriseServiceRequest: serviceRequest
                        using: scContainer
      method properties
      "Total correctness"
      "When this method has completed execution, the
      serviceRequestCategory attribute of the service request object
      will have a value (i.e. the service request will have been
      categorised) and that category will match one of the service
      categories supported by the system."
      true results-in
            methodReturnValue = self ∧
```

```
            serviceRequest serviceRequestCategory notNil ∧
            (scContainer detects:
            ([:each | each serviceQCategory =
            serviceRequest serviceRequestCategory] ifNone: [nil]))
            notNil                    "DL1-01 (CategorisingStrategy)"
      sequential
      self subclassResponsibility
      end-sequential
```

The DefaultCategory subclass is presented next:

```
class DefaultCategory
superclass CategorisingStrategy
instance methods
category modifying
      message pattern categoriseServiceRequest: serviceRequest
                        using: scContainer
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = self ∧
            serviceRequest serviceRequestCategory notNil ∧
            (scContainer detects:
            ([:each | each serviceQCategory =
            serviceRequest serviceRequestCategory] ifNone: [nil]))
            notNil                    "DL1-01(CategorisingStrategy)"
      sequential
      serviceRequest serviceRequestCategory:
      (scContainer first) serviceQCategory
      end-sequential
```

The `p_categorise:using:` method of the **original** ServiceCategoryAllocator class contains the following statement to invoke its own `categoriseServiceRequest:using:` method:

```
      parallel
      categorising := true \+
      self categoriseServiceRequest: (inputQ first) using: scContainer
            if inputQ isEmpty not and: [categorising not]
      end-parallel
```

This method is now modified to invoke the method of the appropriate CategorisingStrategy subclass, as shown below. The new `categorisingAlgorithm` instance variable of the ServiceCategoryAllocator class contains a reference to the instance of the relevant CategorisingStrategy subclass. The way in which this variable is initialized will be explained shortly. The method properties of the modified `p_categorise:using:` method also reflect the new receiver of the `categoriseServiceRequest:using:` message.

```
category cyclic
      message pattern p_categorise: inputQ using: scContainer
      method properties
      "Safe liveness"
      ¬(inputQ isEmpty) ∧ ¬categorising until
            categorising ∧
            categorisingAlgorithm postconditions:
            (#categoriseServiceRequest:using:)
            withArguments: #((inputQ first) scContainer)
                              "DP1-03 (ServiceCategoryAllocator)"
```

```
parallel
categorising := true \+
categorisingAlgorithm categoriseServiceRequest: (inputQ first)
using: scContainer
      if inputQ isEmpty not and: [categorising not]
end-parallel
```

The `categorisingAlgorithm` variable is initialized as follows: When the ServiceCategoryAllocator instance creation method is invoked, the name of the appropriate CategorisingStrategy subclass is passed as an argument. During initialization of the ServiceCategoryAllocator instance, the CategorisingStrategy subclass instance is created and the reference is stored in the `categorisingAlgorithm` variable.

If the CategorisingStrategy subclasses contain parallel methods, then the statements in these methods have to be activated. A **common interface** should be defined for the invocation of the parallel methods of the various subclasses. That would facilitate the activation of the parallel statements of the instantiated subclass by merely **adding a statement to invoke this common method** to the `p_categorise:allocate:` method. The latter is the method which invokes all the parallel methods of the context. It was first introduced in Section 9.5.3. Should the CategorisingStrategy subclasses contain parallel methods, the required modification would be as shown below in bold italics:

```
message pattern p_categorise: inputQ
      allocate: scContainer
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
parallel
self p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] self p_allocate: scContainer from: inputQ
"Once the service request has been categorised, the scAllocator
removes it from the inputQ and appends it to the appropriate
serviceQ."
[] categorisingAlgorithm p_execute
"Execute the parallel statements of the CategorisingStrategy
subclass."
end-parallel
```

From the above it is clear that the Strategy pattern is most suited to a design where such common interfaces can be defined, otherwise it would be more appropriate to subclass the original class.

9.6 Summary

The SLOOP method differs from more conventional object-oriented approaches in the sense that it is based on a **different computational model**. In previous chapters the advantages of this method were described, e.g. its high level of abstraction, its applicability to all types of architectures and its emphasis on correctness properties. In this chapter it was demonstrated that the computational model of the SLOOP method presented no difficulties when applied to such a wide variety of design problems as exemplified by those given in [GHJV95] and [BMRSS96].

Several patterns were incorporated into a design based on the SLOOP method. Each category described in [GHJV95] and [BMRSS] was covered. The suitability of the SLOOP method was discussed for multiple design patterns in each category. The results can be summarized as follows:

- The structure of the **Pipes and Filters architectural pattern** [BMRSS96] promotes parallelism. The filters can be implemented to **execute concurrently** and to be **non-terminating**. These characteristics are **inherent** in the SLOOP approach, since the latter is based on the concept of a number of parallel statements that execute infinitely often.

- The **Reflection architectural pattern** [BMRSS96] was applied during the SLOOP implementation phase. In Chapter 8 it was shown how it could be used to **control statement execution**, perform some **assertion checking** and **generate trace information**.

- The **Factory Method creational design pattern** [GHJV95] allows a class to **defer** the specification of exactly **which** classes to instantiate to its subclasses. This design pattern is easily incorporated into a SLOOP design. Variants that **avoid subclassing** can also be used, but it was pointed out in Section 9.3.1 that this goal can only be achieved successfully if the **parallel** methods of the relevant classes could also be **activated** without requiring subclassing. This is possible if the **client interface** of the parallel methods can be defined to **remain the same for a class and its subclasses**.

- The **Singleton creational design pattern** highlighted another aspect of parallel statement activation. **All the parallel statements** required by a particular application have to be activated via the parallel statements in the *activation-section* of the SLOOP program. However, the instances of some classes may not yet exist immediately after the sequential statements in the *activation-section* have been executed. For example, the class which represents the 'TERMINATING' state of a connection is only instantiated once this state is entered for the first time. The invocation of the parallel methods of such a class therefore has to be **subject to the existence of an instance** of that class. As explained in Section 9.3.2, this is only acceptable if the **netto effect** of the execution of the affected parallel statements is the **same before and after incorporating the design pattern**. An example of how this could be achieved was presented in Section 9.3.2.

- When the **Adapter structural design pattern** is used in a SLOOP design, **no special considerations** are necessary.

- In contrast, care has to be taken that the **mapping** of the collaborators in the **Flyweight structural design pattern** is performed correctly during the implementation phase. A statement selector has to be maintained for **each context** of the shared object in order to guarantee that each parallel statement of the Flyweight will be executed infinitely often for each context.

- One of the applications of the **Proxy structural design pattern** is to make the **physical location** of objects **transparent** to the application. This is in line with the philosophy used in the SLOOP method, which advocates a **unified design approach,** i.e. the target architecture is only considered during the implementation phase.

- The **Iterator behavioural design pattern** is present in Smalltalk library classes. Since **Smalltalk message expressions** may form **part of SLOOP statements**, this design pattern is used extensively in most SLOOP programs.

- The **State behavioural design pattern**, discussed in Section 9.5.2, provided an example of the use of **dynamic** parallel statements. Prior to the incorporation of this design pattern, the parallel statements of the Connection class were **always present in the list of parallel statements**, but they only had an **effect** when the connection was in the 'TERMINATING' state. When these statements were moved to the TerminatingConnection class, they were only **present** in the list of executable statements when the state instance variable of the Connection class referred to the TerminatingConnection class, i.e. when the connection was

in the 'TERMINATING' state. When the application of the State design pattern results in the use of dynamic parallel statements, the designer has to ensure that the **effective** behaviour is the same before and after the incorporation of the pattern.

- The **Template Method behavioural design pattern** is very useful in a SLOOP program. It provides **flexibility** while at the same time it enables the designer to ensure that the **relevant statements will be executed**. This applies to both **sequential and parallel** Template Methods. If the method is **sequential** and **correctness properties** are specified that refer to the **ordering** of the statements, the Template Method allows one to guarantee that these properties will not be violated. Subclasses may only change the **contents** of the methods invoked by the Template Method, but not the invocations themselves.

- In order to use a Template Method design pattern for the invocation of parallel methods, the **client interface** of these methods has to be **the same for a class and its subclasses**. A similar requirement exists when the **Strategy behavioural design pattern** is applied to a SLOOP design.

This chapter concludes the presentation of the various aspects of the SLOOP method. The next chapter summarises the advantages of using this method and it also describes the directions for future research.

CHAPTER 10

CONCLUSIONS

10.1 Evaluation of the SLOOP method

The preceding chapters described all facets of the SLOOP method, viz.

- its syntax,
- the associated semantics,
- the analysis and design approach that results from applying the method,
- reasoning about correctness properties on an informal basis,
- the mapping of a design to an executable program,
- the use of reflection to separate the statements **within** the system being designed from the statements **about** the system being designed, and
- considerations when incorporating various design patterns into SLOOP designs.

Elaborate examples demonstrated various aspects of the SLOOP method. It is now appropriate to evaluate the SLOOP method with respect to the goals of this research as listed in Chapter 1.

10.1.1 Increasing the reliability of systems developed via this method

The first goal, viz. to **maximise** the **reliability** of the system under development, encompasses a wide spectrum of issues. First of all, the system that is produced has to be **functionally correct**. In order to achieve this, the SLOOP method requires the software designer to focus on correctness properties throughout the system development.

During the analysis phase, the **behaviour of the system** is specified in terms of a set of **informal correctness properties,** once the interacting classes have been identified. The SLOOP method aids the designer by providing a **useful checklist**[1] of different kinds of correctness properties that can be specified. This prompts the designer to analyse the problem domain in terms of a wider range of aspects than might otherwise have been the case. This checklist therefore promotes the **completeness** of the specification.

During the design phase, the focus remains on the correctness properties, but now they are **also** used to find suitable **matching** artifacts in the repository of **reusable artifacts** (if one exists). Artifacts are therefore compared on the basis of their correctness properties as was shown in Chapter 6. During the design phase, the correctness properties are **refined**. They are also specified more rigorously in order to facilitate an **unambiguous** specification, which is required in order to **reason about the correctness** of the design. The SLOOP method provides a **notation**[2] based on temporal logic for the rigorous specification of correctness properties.

---

[1] This checklist of correctness properties was described in detail in Chapter 5, Section 5.2.4.
[2] The SLOOP notation for specifying correctness properties was presented in Chapter 4, Section 4.3.4.

During the implementation phase, the target architecture is considered for the first time. When the SLOOP program is mapped to an executable program, the designer has to take care that the semantics of the SLOOP statements are preserved. The issues that need to be taken into account in order to achieve this, were discussed in Chapter 8. The SLOOP method advocates the development of **infrastructures** for mappings to different types of architectures (as described in Chapter 8). The correctness properties of these infrastructures can then also be reused.

It is evident from the above that the SLOOP method provides several mechanisms in order to aid the achievement of **functional** correctness during system development. Functional correctness is a requirement for all types of architectures.

The second aspect of the goal of producing reliable systems is to ensure that the problems usually associated with **concurrency**, such as **deadlock**[3] and **interference**[4], are prevented. Since the SLOOP method advocates a **unified** approach towards software development, these are issues that are only relevant during the implementation phase. The system is designed at a **high level of abstraction**. By definition, the unit of atomic execution in a SLOOP program is a parallel statement. At the design level, all the actions that should take place **atomically**, are grouped into a single parallel statement. There can be no interference between parallel statements.

As described in Chapter 4, Section 4.3.6.4, the SLOOP method is based on the **interleaving model of concurrency** [MaPn81a]. Thus, if two parallel statements refer to the same objects, they execute in some arbitrary order; if they do not share any objects, they may execute simultaneously. During the implementation phase the atomicity of the parallel statements has to be preserved in order to prevent interference. Furthermore, the interleaving model has to be preserved in order to ensure the prevention of deadlock. Chapter 8 covered possible strategies to achieve this.

As was evident from the earlier chapters, deadlock and interference are not the only issues addressed by the SLOOP method. Many different types of **safety**, **liveness** and **precedence** properties are described. Although the software designer is only required to reason about these properties informally, the mere fact that the **"constructive approach"** [Meye90] is followed during system development results in a product that instills more confidence as far as its correctness is concerned. As was demonstrated in Chapter 7, the SLOOP method encourages the software designer to consider both what should happen and also what should never happen. The end result is a more reliable system. This was corroborated by the results of the experimental systems that were developed.

In [Meye97] Bertrand Meyer states that "it is still too difficult to produce software without defects (bugs), and too hard to correct the defects once they are there." He continues to list some of the techniques for improving the reliability of software. These are, *inter alia*, a more **systematic** approach towards software construction, **more formal specifications** and **built-in checks** throughout the software development process.

The SLOOP method applies all of these techniques: It provides a checklist of useful correctness properties, thereby encouraging the software designer to work more **systematically**. The behaviour of the system has to be specified in terms of a set of correctness properties. The SLOOP notation provides the necessary constructs to express these properties **formally**. Although the correctness arguments are informal, they form an integral part of the method, which attests to the significance attached to them. Each phase of the software development process

[3] The conditions for deadlock to occur, as well as deadlock prevention strategies, were described in Chapter 4, Section 4.3.6.5.
[4] Interference was defined in Chapter 2, Section 2.3.2.

emphasizes correctness. By using the SLOOP method, the software developer therefore automatically focusses on correctness issues **throughout** the software development process.

10.1.2 Scalability of the method

When the scalability of the SLOOP method comes under scrutiny, one can argue that there are two aspects that need to be considered when the method is applied to medium- to large-scale systems. The one aspect deals with the software lifecycle in general, i.e. the mechanisms that are provided by the method to handle the **analysis, design and implementation phases**. The second aspect deals with how well the method facilitates **reasoning about correctness**.

When evaluating the way in which the SLOOP method assists the software designer during the analysis, design and implementation phases, the following is apparent. Since the SLOOP method is an object-oriented method, it has all the **structuring capabilities** that are associated with **object-orientation**. Thus, the solution domain is modelled in terms of a set of classes. The SLOOP method takes full advantage of the **data encapsulation** feature of object-orientation. Even the **parallel statements** are defined on a per class basis and are **encapsulated** within parallel methods associated with specific classes. The SLOOP method therefore provides the software designer with the necessary structuring mechanisms in order to break a large system into smaller, more manageable components.

As far as reasoning about correctness is concerned, the SLOOP method has several features that simplify correctness arguments. The benefits are particularly noticeable in larger systems. As demonstrated in Chapter 7, the **correctness arguments do not refer to location counters**. This is because the properties are existentially or universally qualified over all program statements. In larger systems this **reduces the complexity** of the correctness arguments considerably, since in a SLOOP program there is **no need to take computation histories into account**. In a system with a conventional computational model, the number of computation histories grows exponentially as the number of processes that are involved increases.

Location counters are only significant in **sequential methods** in SLOOP programs, but since a sequential method is always executed as an **atomic** unit, there can be no interference and the correctness arguments are therefore as for a sequential program, i.e. relatively simple. Since a sequential method is typically a very **small** piece of code, the software designer only has to deal with a small piece of logic at a time.

Although the correctness properties are quantified over **all** the parallel statements of a SLOOP program, it is typically **only a few** of these statements that actually influence a specific property. Only those parallel statements that reference the objects mentioned in the correctness properties, and possibly a few related ones, need to be taken into account. This was demonstrated in many examples in Chapter 7. Thus, although it might seem that the size of a system would adversely affect the ease with which one could reason about correctness, this is not the case because the **parallel statements** of the system **do not have a flat structure**. Parallel statements are **encapsulated** within the **parallel methods of objects** and are therefore structured according to the classes of the system. Furthermore, the parallel methods of the classes have **correctness properties** associated with them that specify clearly the effects of executing the statements contained within them.

Another feature of the SLOOP method which makes it particularly appropriate for larger systems, is its capacity for **reuse**. Not only can designs and code be reused, but **correctness reasoning** also does **not** need to take place **from first principles** each time. This was demonstrated in numerous examples in Chapter 7. Furthermore, the `postconditions:` and `postconditions:withArguments:` constructs in the **SLOOP notation** make it possible to

highlight the fact that other methods are being invoked and that their correctness properties are being **reused**.

The running example that was used in the body of this thesis was specifically chosen so that the applicability of the SLOOP method to **non-trivial** systems could be demonstrated.

10.1.3 Understandability of the method

A number of issues affect the understandability of the SLOOP method. First of all, the underlying **computational model** has to be understood by the software designer. The user of the SLOOP method has to think in terms of statements that execute infinitely often. Although this is different from conventional computational models, the principle is simple.

The second aspect that a new user of the SLOOP method has to grasp, is how **object-orientation** fits into the picture. Again, this is not complex. The (static) object model of the system is created in the usual way. It is only once the **behaviour** of the system is specified that the computational model has an effect. The designer has to determine how the functionality of a class should be distributed amongst its **sequential** and **parallel** methods. The designer also has to determine which actions should be executed **atomically**, i.e. which actions should be grouped into a single parallel statement.

The fact that the behaviour of the system and its classes is first specified via a set of properties (as was shown in Chapters 5 and 6), makes it relatively easy to derive the SLOOP statements. This is because one has a clear specification of what the behaviour of each class and its methods should be.

One of the main criticisms against formal methods is the perception that the underlying mathematics is difficult and tedious to use. Although the SLOOP method is not a formal method, a certain amount of rigour is required in order to support correctness reasoning. The underlying mathematical foundation for the SLOOP method is based on UNITY [ChMi88]. In [GePn89] a proof system is given for UNITY. It is claimed there that the UNITY assignments could be substituted with arbitrary programs without having to change the existing rules of the proof system; only a few additional rules would have to be added to reason about these atomic programs. This forms the theoretical basis for allowing method invocations in SLOOP statements.

However, **the user is not required to have any detailed knowledge of the theory underpinning the SLOOP method**; not even when reasoning about the correctness of a SLOOP program. As was demonstrated in Chapter 7, the **correctness arguments** are **informal**. There is no need to learn a set of theorems and to understand the application of such theorems. A correctness property is shown to be correct by inspecting the other correctness properties of the program and using them in the correctness arguments. If there are no relevant properties, the SLOOP statements themselves are inspected in order to support the correctness arguments.

Several factors contribute to ensure that this is not such a daunting task. First of all, there is **no need to take location counters into consideration**. Secondly, the **structuring** and the **data encapsulation** provided by object-orientation, results in the **localisation** of the properties (and statements) that need to be considered. It is seldom that all the properties specified for a program need to inspected in order to reason about the correctness of a specific aspect. Furthermore, each correctness property is only proved **once from first principles**. Thereafter its results can be **reused** in other correctness arguments. This reusability contributes towards making the proofs less tedious.

Another important aspect which simplifies the correctness properties and therefore aids understandability, is the fact that **class and instance methods may be used in the correctness**

properties. The only proviso is that they should not modify the state of the objects. One is therefore not restricted to specifying the properties in terms of class and instance variables and boolean operators. The **expressive power** gained from allowing methods in the correctness properties contributes towards making the specification of these properties **less tedious** and it also makes them **more understandable**. It enables one to write these properties in terms of methods rather than in terms of variables, i.e. they are expressed at a **higher level of abstraction**.

In the Eiffel programming language, method invocations are also allowed in the assertions [Meye97]. However, the SLOOP method goes even further. The notation contains the **`postconditions:` and `postconditions:withArguments:` constructs** that enable one to specify which other modifying methods are invoked by the method currently under consideration. This makes it **explicit** that another method is involved. It makes it clear that the postconditions of the other method will hold, without stating what those postconditions are. This **minimises the risk of inconsistency**, since the actual postconditions of that method are specified at one location only, viz. where that method is defined.

Finally, the SLOOP syntax includes **Smalltalk message expressions**, which makes SLOOP programs easily understood by software designers already familiar with Smalltalk.

10.1.4 Unified approach

The SLOOP method is not concerned with the target architecture during the analysis and design phases of system development. This was demonstrated in the example used in Chapters 5 and 6. The **target architecture** is only considered once the **implementation phase** is reached. In Chapter 8 it was shown what issues needed to be considered when a SLOOP program was mapped to various architectures. In all cases the **semantics** of the SLOOP statements, as well as the **atomicity** of the SLOOP parallel statements had to be preserved. This was achieved in different ways for the various architectures, as described in Chapter 8.

Although the mapping to a specific architecture is an additional step that is not required in a software development method where the program is designed for a specific architecture from the outset, a unified approach provides the software designer with the **freedom to map the design to any target architecture** with relative ease. In our experience, there has not been the need to make any modifications to the design as a result of a mapping to a particular target architecture.

A unified approach also has the advantage that the design is at a **high level of abstraction**. There is no need to consider issues such as mutual exclusion and deadlock that would normally be associated with concurrency. Those aspects are addressed during the implementation phase. As shown in Chapter 8, the solutions that are implemented for the various target architectures during that phase can be **reused in infrastructures** for the respective target architectures.

10.1.5 Reusability

The SLOOP method has all the **reusability features** of a **fully-fledged object-oriented method**. In Chapter 9 it was demonstrated how **design patterns** could be reused in a SLOOP design. In addition to the usual reuse of classes and patterns, the SLOOP method also makes provision for the reuse of **correctness properties**, as was described in Chapter 7. During the implementation phase, the **mapping infrastructures** can also be reused. This was discussed in detail in Chapter 8.

10.1.6 Seamlessness

During the analysis phase, the behaviour of the system under development is described in terms of a set of correctness properties. During the design phase, these properties are refined and

additional design level properties are added. The notation used for the specification of the properties includes Smalltalk message expressions. This makes the derivation of the SLOOP statements from the correctness properties relatively simple. If the target implementation language is Smalltalk-80, the transition from the design phase to the implementation phase is **seamless**.

As demonstrated in Chapter 8, it is possible to use the **reflective facilities** of Smalltalk to ensure that the base objects do not contain any additional logic that are specific to the mapping of the program to the target architecture. For example, the statements related to the selection of the next parallel statement for execution is relegated to the metaclasses. The mapped base class and the original SLOOP class are therefore almost identical, thereby achieving a high degree of seamlessness.

10.1.7 General availability and minimisation of developmental resources

At this stage a SLOOP development environment does not exist yet. Currently, a SLOOP program can be written using any text editor. When the mapping is performed, the development environment of the target architecture is used. For example, for a mapping to a sequential architecture the Smalltalk-80 development environment can be used as is. This also applies to a mapping to an asynchronous shared-memory architecture where multiple processes run on a single processor. The meta-object infrastructure used for reflective computation and the mapping infrastructures are implemented as reusable classes. All of these factors make it easy to experiment with the concepts proposed in the SLOOP method without having to make a large investment in terms of developmental resources.

10.2 Concluding remarks and future research directions

The purpose of this research has been to **experiment** with the concept of an **object-oriented method based on a Single Location Program (SLP) computational model** in order to try and achieve the goals discussed in the previous section. As recorded in the preceding chapters and summarised in the previous section, the SLOOP method provides the necessary features to accomplish this.

This experiment has therefore **resulted** in the development of a **new software construction method** which has the rich feature set of an object-oriented method, but which is based on a computational model that simplifies correctness reasoning. This **simplification**, which is enhanced by the high degree of **reuse** that is facilitated by the object-oriented nature of the method, has the following implications:

- It improves the understandability of the method.
- It makes the specification of correctness properties a simpler and less tedious task.
- Informal correctness reasoning about these properties becomes viable.
- It makes the method attractive to practising software designers that are not necessarily proficient in the use of formal methods.

The SLOOP method therefore promotes a **"constructive approach"** towards software development.

Although the emphasis in this research has been on creating a software development which facilitates **informal** correctness reasoning, this could be **complemented** by the development of a formal proof system for the SLOOP method. That way, the method would still be usable without requiring any knowledge of formal methods if only informal correctness arguments were used, but the user would also have the option of creating formal proofs. **Further research** would be required to develop a **formal semantics and proof system** for the SLOOP method. That would

then also facilitate the development of **tools** to **automate** the verification of programs developed via the SLOOP method.

Although the mapping to an existing executable language such as Smalltalk is straightforward, the purpose of this work was **not** to **maximise the efficiency of the executable program**. That is a topic for further research. Aspects that are currently implemented via reflection in metaclasses should form part of the **development environment**. A **SLOOP translator** could form part of such a development environment.

It must be noted that the choice of Smalltalk as the language to provide the SLOOP method with its object-oriented facilities was motivated to a large extent by the suitability of Smalltalk for experimental systems (this includes its reflective facilities). There is no reason why another object-oriented language could not replace Smalltalk in the SLOOP method when a fully fledged development environment is developed. However, such a language would have to provide at least the same capabilities as Smalltalk (except for the reflective facilities).

Desirable features of such a language would be proper support for **encapsulation** (in the case of C++ and Java this is somewhat illusory [ABV00]), **polymorphism** and **inheritance**. If further research shows that the SLOOP method should support **multiple inheritance**, then that would be a desirable feature of such a language as well. **Strong typing** would facilitate the detection of certain errors[5] during compilation and it would also be possible to implement compilation optimisations that could increase efficiency [Meye97]. A strongly typed language would then require the support for **genericity**, i.e. classes with formal generic parameters representing arbitrary types would have to be supported [Meye97].

As far as **software development tools** are concerned it would be very useful to have an **animated graphical trace facility**. Such a tool would be used during the design phase to generate animated execution traces. It would highlight an object on the screen whenever that object executes an unconditional parallel statement or whenever it executes a conditional parallel statement and the condition evaluates to true. The values of some of the instance variables as they are after the execution of the parallel statement could be shown. If sequential methods are invoked by the parallel statement, then the target objects could be highlighted in another colour. Such a tool would not replace the correctness reasoning described in Chapter 7. However, it could be used to aid **understandability**, since it would facilitate the **visualisation of event flows**.

Another aspect that could be investigated further is to find **more succinct** ways of presenting the informal correctness arguments, **without** going to the lengths of changing SLOOP into a formal method. The inclusion of **real time properties** into the formalism is another potential area for further investigation.

This research has produced very encouraging results. The SLOOP method facilitates solutions that are elegant, reusable, extendible, understandable and reliable. Further research would enhance the method, but it can already be applied successfully in its existing form. A solid foundation has been laid for creating high quality software systems.

[5] It would be possible to detect during compilation that a message is being sent to an object which does not implement that message.

APPENDIX A

SLOOP SYNTAX QUICK REFERENCE

A.1 Notational conventions

The SLOOP syntax is given below in BNF. The significance of the typeface and symbols used in the BNF description is as follows:

Plain or boldface type	Terminal symbols
Italics	Non-terminal symbols
Braces	Grouping
Square brackets	Enclosed syntactical unit is optional
Asterisk	Zero or more occurrences of the syntactical unit
Plus	One or more occurrences of the syntactical unit
Vertical bar	Separates options
Single quotes	Enclose literals.

All names, such as *program-name*, *class-name*, *instance-name* and *category-name*, are strings that may contain letters, digits and the underscore character. They all start with a letter. If the plural form is used (for example as in *instance-variable-names*), then one or more name(s) may be present, each separated by a white-space character. Three consecutive colons separate a *class-name* from a *package-name* when it is necessary to qualify the *class-name* by a *package-name*.

A.2 The SLOOP program structure

A *SLOOP-program* has the following structure :

SLOOP-program → **program** *program-name*
activation-section
{*package-description*}+
end-program

activation-section → **sequential**
statement-list
end-sequential
parallel
statement-list
end-parallel

package-description → **package** *package-name*
{*package-description*}*
{*class-description*}*
{*partial-class-description*}*
end-package

A.3 Complete and partial class descriptions

Syntax of a *class-description*:

class-description →
class *class-name*
superclass *superclass-name* [**from** *repository-name*]
[**class variable names** [*class-variable-names*]]
[**instance variable names** [*instance-variable-names*]]
[**class macros** [*macros-section*]]
class properties [*properties-section*]
methods-section

Syntax of a *partial-class-description*:

partial-class-description →
class *class-name* **from** *repository-name*
partial-class-methods-section

The **class properties** keywords are mandatory in order to indicate to the designer that all the relevant properties should always be listed. If this section had been optional, then it would have been unclear to the person reusing the class whether the original designer had merely chosen not to list the properties or whether there had been no relevant properties to list.

A.4 The *macros-section*

Syntax of a *macros-section*:

macros-section → *macro-list*
macro-list → *macro-definition*
{[] *macro-definition*}*
macro-definition → *macro-variable* ≡ *macro-expression*
macro-variable → *variable-name*
macro-expression → *simple-macro-expression* |
conditional-macro-expression
simple-macro-expression → *message-expression* | *variable-name* | *literal*
conditional-macro-expression → *simple-macro-expression*
if *boolean-expression*
{~ *simple-macro-expression*
if *boolean-expression*}*

A *message-expression* is a Smalltalk-style message expression and a *boolean-expression* is a Smalltalk-style message expression that returns true or false. A Smalltalk-style message expression consists of a *receiver*, a *selector* and zero or more *arguments*. If there are no arguments, the message is called a *unary message*. For example, `bufferedElements size` is

a unary message expression. A *binary message* has a single argument following a selector consisting of one or two non-alphanumeric characters, the second of which may not be a minus sign. The message expression `a + b` is an example of a binary message expression. The third type of message is a *keyword message*. The selector consists of one or more keywords, each with its associated argument. A keyword consists of an identifier followed by a colon. For example, `bufferedElements addLast: newElement` is a keyword message expression.

A *literal* is a Smalltalk-style literal which may be a number, a symbol constant, a character constant, a string or an array constant.

Note that, as in Smalltalk-80, message expressions may be nested. The receiver of a message expression may itself be a message expression. Similarly, the argument(s) of a keyword message may also be message expression(s).

A message expression may also contain a block. The reader is referred to [GoRo89] for a detailed discussion of a Smalltalk-80 block. For the purposes of its application in SLOOP, the following description suffices.

A block represents a deferred sequence of actions. It consists of a sequence of expressions separated by periods and delimited by square brackets. The actions represented by a block are not necessarily executed when the block expression is encountered. For example, a block expression is used as the argument of the Smalltalk-80 `and:` keyword message. This message represents the logical "and" operation. If the first operand (i.e. the receiver) evaluates to true, the second operand (the argument of `and:`) is evaluated and its value is returned as result. However, if the first operand evaluates to false, the second operand is not evaluated. This is indicated syntactically via the fact that theargument of the `and:` message is a block expression.

A block is also used when the receiver of a message is a collection and the actions represented by the block need to be applied to each element of the receiver. In that case each element of the receiver is passed to the block as an argument. This is indicated syntactically by the presence of an identifier preceded by a colon at the beginning of the block. This identifier is separated from the rest of the contents of the block by a vertical bar.

For example, the Smalltalk-80 library `select:` and `detect:` messages are used frequently in the CallCentreSimulation example. The `select:` message evaluates the block received as argument of the message for each of the receiver's elements (the receiver is a collection object). It returns a collection that contains only those elements of the receiver for which the block evaluates to true. The message expression below returns the collection representing all employees earning a salary greater than $20 000:

```
employees select: [:each | each salary > 20000]
```

The following message expression returns the object representing the first employee found earning a salary greater than $20 000 (if no such employee is found, then nil is returned):

```
employees detect: [:each | each salary > 20000] ifNone: [nil]
```

For a formal description of the Smalltalk-80 syntax, the reader is referred to [GoRo89].

A.5 The *properties-section*

Each class, as well as each method within a class may contain a *properties-section*. The syntax is as follows:

properties-section → *property-list*
properties-list → *property*
{[] *property*}*

A *property* may be of the form:
p **unless** *q*,
stable *p*,
invariant *p*,
p **ensures** *q*,
p **leads-to** *q*,
p **until** *q*,
p **detects** *q*,
p **precedes** *q* or
p **results-in** *q*,
where *p* and *q* are **first-order** predicates.

Since **first order predicate logic** is used, universal and existential quantification is allowed in the logical relations. The keywords **forall** and **exists** may be used as alternatives to the ∀ (universal quantification) and ∃ (existential quantification) symbols respectively. Instead of using a colon to denote the domain of a quantification, the reserved word **where** is used. This is to avoid confusion with the colon used in Smalltalk keyword expressions.

If a variable-list is used in a quantification, the variables are separated by a comma preceded by a backslash. This ensures that the comma cannot be mistaken for the Smalltalk concatenation symbol. If a '<' symbol is followed immediately by a quantification symbol, it denotes the start of a quantification construct. If a '>' symbol appears as the first non-white-space character on a line, it denotes the end of a quantification construct. The quantification constructs have the same parsing precedence as parentheses.

Smalltalk-style message expressions may be used in the predicates. However, these message expressions may not have any side-effects, i.e. they may not change the state of any object.

The ∧ (logical and), ∨ (logical or) and ¬ (negation) operators are defined in addition to the Smalltalk & (logical and), | (logical or) and not (negation) operators. The additional logical operators serve a readability purpose only. One difference between the additional logical operators and the Smalltalk ones is in the parsing precedence. The Smalltalk unary, binary and keyword expressions are evaluated in that order. The additional logical operators are evaluated after the unary, binary and keyword expressions have been evaluated.

Further conventions about the priorities of logical relations are given next (those on the same line have equal priority and the lines represent the priorities from high to low):
¬
=, ≠
∧, ∨
⇒
≡
unless, ensures, leads-to, stable, invariant, detects, until, precedes

Properties are universally quantified over all the free variables occurring in them.

A.6 The *methods-section*

The syntax of the *methods-section* of a class is as follows:

methods-section → [**class methods**
{*methods-implementation*}+]
[**instance methods**
{*methods-implementation*}+]

methods-implementation → **category** *category-name*
{*method-description*}+

method-description → *sequential-method* | *parallel-method*

sequential-method → **message pattern** *Smalltalk-message-pattern*
[**method macros** *macros-section*]
method properties *properties-section*
sequential
statement-list
end-sequential

parallel-method → **message pattern** p_*Smalltalk-message-pattern*
[**method macros** *macros-section*]
method properties *properties-section*
parallel
statement-list
end-parallel

partial-class-methods-section →
[**class methods**
{*selector*}+]
[**instance methods**
{*selector*}+]

selector → *Smalltalk-selector* |
p_*Smalltalk-selector*

A *Smalltalk-message-pattern* has the usual Smalltalk syntax, i.e. it comprises the message selector with the associated pseudo-variables to represent the arguments if there are any. A *Smalltalk-selector* has the usual syntax of a selector in a Smalltalk program.

Similarly to the properties of the class, all the relevant properties of the method should be listed if there are any. If this section had been optional, then it would have been unclear to the person reusing the class whether the original designer had merely chosen not to list the properties or whether there had been no relevant properties to list.

A.7 SLOOP statements

The SLOOP *statement-list* is defined as follows:

statement-list → *statement*[1]
{[] *statement* }*

statement → *simple-statement* |
quantified-statement-list

quantified-statement-list → <[] *quantification statement-list*>

[1] This implies that, as in UNITY, a statement-list cannot be empty.

quantification	→	*variable-list* **where** *boolean-expr* ::
variable-list	→	*variable* {\, *variable*}*
simple-statement	→	*statement-component* { || *statement-component* }*
statement-component	→	*enumerated-component* | *quantified-component*
enumerated-component	→	*component-part* | *conditional-component-part-list*
quantified-component	→	<|| *quantification simple-statement*>
component-part	→	{[^]*variable* := *simple-expr*}[2] | [^]*message-expression*
conditional-component-part-list		
	→	*simple-component-part-list* if *boolean-expr* {~ *simple-component-part-list* if *boolean-expr* }*
simple-component-part-list	→	*component-part* {\+*component-part*}*
simple-expr	→	*message-expression* | *primary*

A *message-expression* is a Smalltalk-style message expression and a *boolean-expr* is a Smalltalk-style message expression that returns true or false. A Smalltalk-style message expression consists of a receiver, a selector and zero or more arguments. If there are no arguments, the message is called a unary message. A binary message has a single argument following a selector consisting of one or two non-alphanumeric characters, the second of which may not be a minus sign. The third type of message is a keyword message. The selector consists of one or more keywords, each with its associated argument. A keyword consists of an identifier followed by a colon.

A *primary* is a Smalltalk-style primary which may be a variable name, a literal or a block. When a "<" symbol is immediately followed by the "[]" or "||" symbol, it denotes a quantification and the "<" symbol is not interpreted as a Smalltalk operator. If a ">" symbol appears as the first non-white-space character on a line, it denotes the end of a quantification construct.

No messages related to the Smalltalk-80 support for multiple processes may be used, since there is no concept of a process in a SLOOP program.

Cascaded message expressions are also not allowed. The motivation for this restriction was given in Chapter 4, Section 4.3.6.2.

A.8 Comments in a SLOOP program

Comments may be inserted anywhere in the program and are enclosed by double quotes.

[2] The braces around the [^]*variable* := *simple-expr* construct serve to identify the latter as a syntactic unit. This is needed because the '|' symbol has a higher precedence than the ':=' symbol.

APPENDIX B

A SLOOP PROGRAM FOR A CALL CENTRE

B.1 Scope of the first level of refinement of the design

The first level of refinement is only concerned with normal behaviour; no error conditions are specified. This implies that a service user is always served once it is connected to the system; the possibility of aborting the connection (hanging up before the service has been completed) or rejecting the service request (e.g. due to unavailability of the relevant service providers) is not specified at this stage.

This is a cyclic system, which should therefore not terminate. However, there may be conditions under which the system may need to be shut down. System shutdown is not shown at this level of refinement.

The service providers may only be in the 'BUSY' or 'IDLE' states. At this level of refinement there is no `state` instance variable. The 'BUSY' and 'IDLE' status of a service provider is determined by checking whether it can accept the next service request or not. The 'RESTRICTED-IDLE', 'RESTRICTED-BUSY' and 'UNAVAILABLE' states are introduced during subsequent refinements. The additional states enable a service provider to go out of service gracefully and it ensures that once a service request is allocated to a service queue that it will be serviced.

(When a service provider is operating in a restricted mode, it means that it will continue to remain active until all the service requests already present in the service queue at the time when it changes to the restricted mode have been serviced. This indicates to the `scAllocator` that no further service requests should be accepted for a specific service queue if all the service providers servicing that particular category are operating in a restricted fashion.)

At this level of refinement the service user is not informed of the progress of the service request.

An object diagram of the Call Centre system is shown in Figure B-1. Figure B-2 illustrates the contents of the various packages in the system.

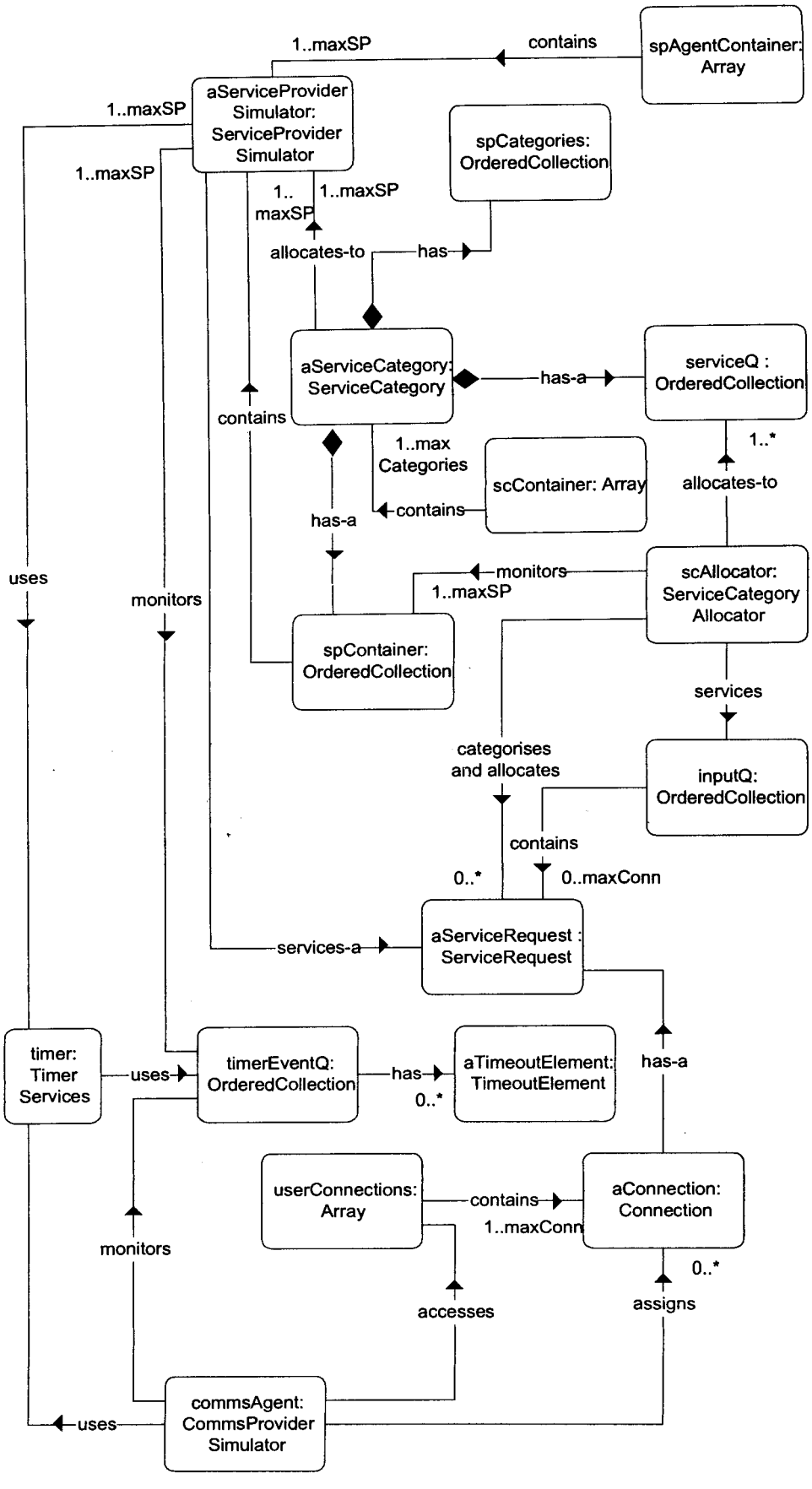

Figure B-1. Call centre object diagram.

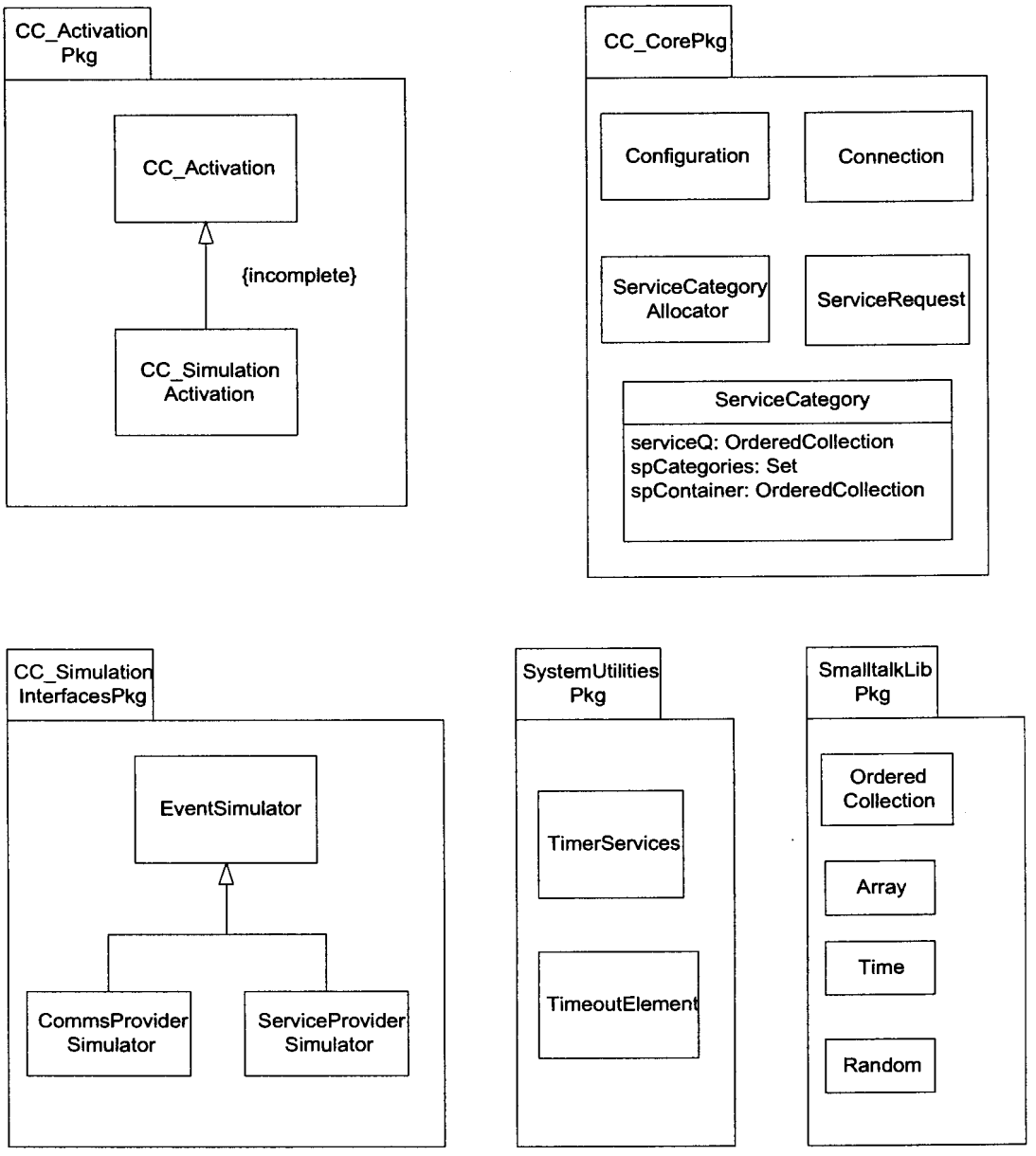

Figure B-2. Contents of the various call centre packages.

Full class descriptions are presented here for all non-Smalltalk library classes.

The property numbers are defined uniquely with respect to the class in which they appear. These numbers may appear in any order as long as they are unique. These numbers are independent of the numbers of the superclass properties. If a property overrides a property specified in an ancestor, the number of the ancestor property is used in the subclass followed by the name of the ancestor in brackets. When referring to a property that overrides a property in its superclass, the property identifier as described above must be followed by the words in *class-name*, where *class-name* is the name of the subclass.

In order to demonstrate that the concept of a package has no significance other than organizational, the classes in this Appendix are presented as individual classes and in any order.

B.2 The CC_Activation class

The CC_Activation class is an abstract superclass which leaves the activation of the **interface** classes up to the subclasses. It activates all the other classes in the system. The CC_SimulationActivation class overrides the methods related to the interface classes. The analysis level properties are not repeated here. They can be found in Chapter 5, Section 5.4.

```
class CC_Activation
"The CC_Activation instance instantiates all the classes of the system
that need to present before the parallel methods start executing.  It
also activates all the parallel methods required by the system."

superclass SmalltalkLibPkg:::Object from SmalltalkLibRepository
instance variable names
"The following variables represent the objects that are not
instantiated as elements of a collection, i.e. they are not indexed
instance variables."
config
      "All configurable values (e.g. the maximum number of connections
      supported) are obtained via the config object."
commsAgent
      "The commsAgent handles the interface with the communication
      provider."
userConnections
      "This variable represents the collection of connections
      supported by the system."
inputQ
      "The inputQ models the FIFO way in which service requests are
      accepted by the system."
scAllocator
      "The scAllocator categorises the service requests and allocates
      them to the appropriate service queues."
scContainer
      "This variable represents the collection of service categories
      supported by the system."
spAgentContainer
      "The spAgentContainer contains all the service providers
      supported by the system."
timer
      "The timer object provides timer services to the other objects
      in the system."
timerEventQ
      "The timerEventQ is used to inform the requestors of the various
      timeouts that a requested timeout has occurred."

class macros
      maxConn ≡ config maximumConnections
            "Number of simultaneous user connections supported"
      [] maxCategories ≡ config maximumServiceCategories
            "Number of service categories supported"
      [] maxSP ≡ config maximumServiceProviders
            "Number of service providers supported"

class properties
"These are the properties as identified during the analysis phase, as
well as the following design level clean behaviour invariants:"
      invariant
      config notNil ∧
      commsAgent notNil ∧
      userConnections notNil ∧
```

```
        < ∀ i where 1 ≤ i ≤ maxConn :: (userConnections at: i) notNil
        > ∧
        inputQ notNil ∧
        scAllocator notNil ∧
        scContainer notNil ∧
        < ∀ j where 1 ≤ j ≤ maxCategories ::
              (scContainer at: j) notNil
        > ∧
        spAgentContainer notNil ∧
        < ∀ k where 1 ≤ k ≤ maxSP :: (spAgentContainer at: k) notNil
        > ∧
        timer notNil ∧
        timerEventQ notNil ∧
        maxConn > 0 ∧
        maxCategories > 0 ∧
        maxSP > 0                                                    "DS2-01"
        "Clean behaviour"
        "The CC_Activation class is an abstract class and should not be
        instantiated"
        <∀ anObject :: invariant anObject class ~~ CC_Activation
        >                                                            "DS2-02"

instance methods
category private
      message pattern initialize
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = self ∧
            config notNil ∧
            self postconditions: (#initManagement) ∧
            commsAgent notNil ∧
            self postconditions: (#initCommsAgent) ∧
            userConnections notNil ∧
            < ∀ i where 1≤i≤maxConn :: (userConnections at: i )
            notNil
            > ∧
            inputQ notNil ∧
            scAllocator notNil ∧
            self postconditions: (#initServiceCategoryAllocator) ∧
            scContainer notNil ∧
            < ∀ j where 1 ≤ j ≤ maxCategories :: (scContainer at: j)
            notNil
            > ∧
            spAgentContainer notNil ∧
            < ∀ k where 1 ≤ k ≤ maxSP :: (spAgentContainer at: k)
            notNil
            > ∧
            timer notNil ∧
            (timer class) postconditions:(#setup:)
            withArguments: #(config) ∧
            timerEventQ notNil                                       "DL1-02"
      "Note that the receiver of the postconditions:withArguments:
      message is the expression (timer class) instead of
      SystemUtilitiesPkg:::TimerServices. This is done to facilitate
      subclassing without violating the correctness properties. If
      the actual class name had been used here, then the property
      would no longer have been valid if a subclass of TimerServices
```

had been instantiated at this point. Recall that correctness properties must be preserved during subclassing."

"At this stage (i.e. before the subclass has completed the execution of its instance creation method) the class invariants do not need to hold yet, so it should be stated explicitly that once the predicate *self postconditions: (#initManagement)* holds, it continues to hold. That is a requirement, since many of the subsequent statements in the method depend on it. Similarly for the other **stable** properties listed below."

```
stable config notNil ∧ self postconditions: (#initManagement)
                                                    "DS4-01"
```

"Note that self postconditions: (#initManagement) implies that:

```
maxConn > 0 ∧
maxCategories > 0 ∧
maxSP > 0 "

stable userConnections notNil                       "DS4-02"
stable scContainer                                  "DS4-03"
stable spAgentContainer notNil                      "DS4-04"

sequential
config := self initManagement
[] commsAgent := self initCommsAgent
[] userConnections := SmalltalkLibPkg:::Array new: maxConn
[] < [] i where 1≤i≤maxConn :: userConnections at: i
      put: (self initConnection: i)
  >
[] inputQ := SmalltalkLibPkg:::OrderedCollection new: maxConn
[] scAllocator := self initServiceCategoryAllocator
[] scContainer := SmalltalkLibPkg:::Array
    new: maxCategories
[] < [] j where 1≤j≤maxCategories :: scContainer at: j
    put: (CC_CorePkg:::ServiceCategory setup: config)
  >
[] spAgentContainer :=
      SmalltalkLibPkg:::Array new: maxSP
[] < [] k where 1≤k≤maxSP :: spAgentContainer at: k
    put: (self initSPAgent)
  >
[] timer:= SystemUtilitiesPkg:::TimerServices setup: config
[] timerEventQ := SmalltalkLibPkg:::OrderedCollection new
end-sequential


message pattern initManagement
method properties
"Total correctness"
true results-in
      methodReturnValue notNil ∧
      (methodReturnValue class) postconditions:(#setup)
                                                    "DL1-03"
      "Again the explicit reference to a class name (in this
      case CC_CorePkg:::Configuration) is avoided in order to
      ensure that subclasses do not violate the correctness
      property."
sequential
^CC_CorePkg:::Configuration setup
end-sequential
```

```
        message pattern initConnection: index
        method properties
        "Total correctness"
        true results-in
              methodReturnValue notNil ∧
              (methodReturnValue class) postconditions:(#setup:)
              withArguments: #(index)                              "DL1-04"
        "Again the explicit reference to a class name (in this case
        CC_CorePkg:::Connection) is avoided."
        sequential
        ^CC_CorePkg:::Connection setup: index
        end-sequential

        message pattern initCommsAgent
        method properties
        "Total correctness"
        true results-in methodReturnValue notNil                   "DL1-05"
        sequential
        self subclassResponsibility
        end-sequential

        message pattern initSPAgent
        method properties
        "Total correctness"
        true results-in methodReturnValue notNil                   "DL1-06"
        sequential
        self subclassResponsibility
        end-sequential

        message pattern initServiceCategoryAllocator
        method properties
        "Total correctness"
        true results-in methodReturnValue notNil                   "DL1-07"
        sequential
        ^CC_CorePkg:::ServiceCategoryAllocator setup
        end-sequential

category cyclic
        message pattern p_activate
        method properties
        "These are the properties as identified during the analysis
        phase."
        "The p_activate method is only invoked once the CC_Activation
        subclass has been instantiated.  The class invariants of the
        CC_Activation subclass that has been instantiated are therefore
        guaranteed to hold before the p_activate method is executed.
        Each statement executed by the p_activate method has to preserve
        these invariants."


        parallel
        self p_executeCPAgent
        "The parallel methods of the commsAgent are not invoked
        directly, but rather via the p_executeCPAgent method of the
        CC_Activation class."

        [] timer p_runTimer: timerEventQ
        "Activate the parallel methods of the timer object.  The timer
        parallel statements have the following functionality: Whenever a
        timeout occurs, the TimeoutElement instance representing the
        timeout is added to the end of the timerEventQ, which indicates
        to the requestor that the specified timer has expired."
```

```
[] self p_categoriseAndAllocate
"The parallel methods of the scAllocator object are invoked via
the p_categoriseAndAllocate method of the CC_Activation class.
The scAllocator parallel statements have the following
functionality: Once a service request has been categorised, it
is removed from the inputQ and appended to the appropriate
serviceQ."

[] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
            p_execute
  >
"Activate the parallel methods of the ServiceCategory instances.
Their parallel statements have the following functionality: For
each service category the associated service queue and set of
service provider agents are monitored.  If the service queue is
not empty and a service provider agent in the spSubset
associated with the service category is available to process a
new service request, the first element of the service queue is
removed and assigned to a service provider agent."

[] < [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
  >
"The p_executeConnection method of the CC_Activation class is
executed for each Connection instance in order to invoke the
parallel methods of the latter.  The parallel statements of the
Connection instances have the following functionality: When a
connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection.  Once all the procedures have been completed to
terminate the connection, the connection and its associated
service request are reset to their initial states."

[] < [] k where 1≤k≤maxSP :: self p_executeSPAgent:
      (spAgentContainer at: k)
  >
"The parallel methods of the service provider agents are not
invoked directly, but rather by executing the p_executeSPAgent
method of the CC_Activation class for each of the service
provider agents."

end-parallel


message pattern p_executeCPAgent
method properties
"These are the properties pertaining to the communication
provider interface as identified during the analysis phase."
parallel
self subclassResponsibility
end-parallel

message pattern p_executeSPAgent: spAgent
method properties
"These are the properties pertaining to the service provider
interface as identified during the analysis phase."
parallel
self subclassResponsibility
end-parallel

message pattern p_categoriseAndAllocate
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
```

```
parallel
scAllocator p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] scAllocator p_allocate: scContainer from: inputQ
"Once the service request has been categorised, the scAllocator
removes it from the inputQ and appends it to the appropriate
serviceQ."
end-parallel

message pattern p_executeConnection: aConnection
method properties
"These are the properties pertaining to the Connection class as
identified during the analysis phase."
parallel
aConnection p_informCommsProvider: commsAgent
"When a connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection."
[] aConnection p_doWrapUp
"Once all the procedures have been completed to terminate a
connection, the connection and its associated service request
are reset to their initial states."
end-parallel
```

B.3 The CC_SimulationActivation class

```
class CC_SimulationActivation
superclass CC_Activation
class properties
class methods
category instance creation
      message pattern setup
      method properties
      "Total correctness:
      After the statements in the initialize method have been executed
      the clean behaviour invariant of the CC_Activation class will
      hold, i.e. all the objects that should be created upon start-up
      will exist.  This implies that the invariants of the classes
      instantiated by the CC_Activation class will also hold, as well
      as the correctness properties of their respective creation
      methods.  All classes have to preserve their respective class
      invariants after initialisation."
      <∀ k where k ≥ 0 ::
      self instanceCount = k results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      ^super new initialize
      end-sequential

instance methods
category private
      message pattern initCommsAgent
      method properties
      "Total correctness"
      true results-in
            methodReturnValue notNil ∧
            CC_SimulationInterfacesPkg:::CommsProviderSimulator
            postconditions: (#startSimulation)
                                          "DL1-05 (CC_Activation)"
      sequential
      ^CC_SimulationInterfacesPkg:::CommsProviderSimulator
            startSimulation
      end-sequential

      message pattern initSPAgent
      method properties
      "Total correctness"
      "The references to scContainer in the precondition of this
      method are required, because during initialization the
      ServiceProviderSimulator instance registers itself with each
      ServiceCategory instance that is serviced by a service provider
      category matching that of the current ServiceProviderSimulator
      instance."
      config notNil ∧ self postconditions: (#initManagement) ∧
      scContainer notNil ∧
      <∀ j where 1 ≤ j ≤ maxCategories :: (scContainer at: j) notNil
      > results-in
            methodReturnValue notNil ∧
            CC_SimulationInterfacesPkg:::ServiceProviderSimulator
            postconditions: (#startSimulation:using:) withArguments:
            #(scContainer config)
                                    "DL1-6 (CC_Activation)"
```

```
        sequential
            ^CC_SimulationInterfacesPkg:::ServiceProviderSimulator
            startSimulation: scContainer using: config
        end-sequential


category cyclic
        message pattern p_executeCPAgent
        method properties
        "These are the properties pertaining to the communication
        provider interface as identified during the analysis phase."
        parallel
        commsAgent p_simulate: timer timeoutEventsIn: timerEventQ
        [] commsAgent p_generateEvent: userConnections target: inputQ
        "The commsAgent simulates the establishment of new connections
        at random intervals (within a configured range).  A simulation
        timer is started after initialization and restarted each time
        after the establishment of a connection has been simulated. The
        latter is done by placing the service request associated with
        the new connection into the input queue.  The commsAgent ensures
        that the capacity of maxConn connections per call centre is not
        exceeded, therefore a message is displayed indicating that all
        connections are busy if the maximum number of connections are
        currently assigned."
        end-parallel

        message pattern p_executeSPAgent: spAgent
        method properties
        "These are the properties pertaining to the service provider
        interface as identified during the analysis phase."
        parallel
        spAgent p_simulate: timer timeoutEventsIn: timerEventQ
        "When a service request has been assigned to a service provider
        simulator, the latter simulates the time it takes to service the
        service request by starting a random timer.  When this timer
        expires, it represents the completion of the service."
        [] spAgent p_generateEvent
        "When the service provider has completed the service, it
        indicates that the connection should be terminated."
        [] spAgent p_updateCategoryIndex: scContainer
        "Update the index into the categoriesServed collection if the
        serviceQ of the current category being served by this spAgent is
        empty."
        end-parallel
```

B.4 The Configuration class

class Configuration
"The purpose of this class is to ensure that the following parameters are configured:

Uses defaults to set the maximum number of connections, service categories and service providers. It also facilitates the configuration of the maximum allowable timeout value, the service request category names supported by the system, the service provider category names supported by the system and the mapping of service request to service provider category names. Subclasses may allow the operator to specify other values."

superclass Object
instance variable names
maximumConnections
"The maximum number of connections supported by the system."
maximumServiceCategories
"The number of service categories supported by the system."
maximumServiceProviders
"The number of service providers supported by the system."
maximumAllowableTimeout
"The maximum allowable timeout that may be requested by any object in the system."
srCategoryNames
"The collection of service request category names supported by the system."
spCategoryNames
"The collection of service provider category names supported by the system."
srToSpCategoryMap
"The mapping of service request categories to service provider categories."
categoriesAssigned
"This variable is used to keep track of the number of service request category names that have already been assigned. When a ServiceCategory instance is created and initialized, it obtains the name of the service category that it supports from the Configuration instance (via the assignSRCategory method). Each service request category name may only be assigned once (in order to ensure that each ServiceCategory instance will support a unique service request category)."

class properties
<∀ (t, u, v, w) **where**
t > 0 ∧ u > 0 ∧ v > 0 ∧ w > 0 ::
invariant
maximumConnections = t ∧
maximumServiceCategories = u ∧
maximumServiceProviders = v ∧
maximumAllowableTimeout = w
> **"DS2-01"**
"The values of each of the maximumConnections, maximumServiceCategories, maximumServiceProviders and maximumAllowableTimeout instance variables are invariant and always greater than zero."

```
<∀ u where u > 0 ::
invariant
      srCategoryNames notNil ∧ srCategoryNames size = u
>                                                          "DS3-01"
```

"The number of service request category names that are configured is equal to maximumServiceCategories."

```
<∀ ( anSRCategoryNameX, anSRCategoryNameY) where
srCategoryNames includes: anSRCategoryNameX ∧
srCategoryNames includes: anSRCategoryNameY ::
invariant
      anSRCategoryNameX ~~ anSRCategoryNameY
>                                                          "DS3-02"
```

"Each configured service request category name is unique."

```
invariant
      spCategoryNames notNil ∧ ¬spCategoryNames isEmpty
                                                           "DS3-03"
```

"At least one service provider category name is configured."

```
<∀ ( anSPCategoryNameX, anSPCategoryNameY) where
spCategoryNames includes: anSPCategoryNameX ∧
spCategoryNames includes: anSPCategoryNameY ::
invariant
      anSPCategoryNameX ~~ anSPCategoryNameY
>                                                          "DS3-04"
```

```
class methods
category instance creation
      message pattern setup
      method properties
      "A Configuration instance is created and initialized.  The new
      instance is returned"

      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      ^super new configure
      end-sequential


instance methods
category private
      message pattern configure
      method properties
      "Upon completion of the configure method, the
      maximumConnections, maximumServiceCategories, maximumService=
      Providers and maximumAllowableTimeout instance variables will
      each have a value greater than zero, the srCategoryNames and
      spCategoryNames collections will have been created, the number
      of elements in the srCategoryNames collection will be equal to
      maximumServiceCategories, there will be at least one element in
      the spCategoryNames collection, the srToSpCategoryMap will have
      been created, the number of mappings in this collection will be
      equal to maximumServiceCategories and the categoriesAssigned
      variable will have the value zero."
```

```
        "Total correctness"
        <∀ ( t, u, v, w) where
        t > 0 ∧ u > 0 ∧ v > 0 ∧ w > 0 ::
        true results-in
              maximumConnections = t ∧
              maximumServiceCategories = u ∧
              maximumServiceProviders = v ∧
              maximumAllowableTimeout = w ∧
              srCategoryNames notNil ∧ spCategoryNames notNil ∧
              srCategoryNames size = u ∧
              ¬ spCategoryNames isEmpty ∧
              srToSpCategoryMap notNil ∧
              srToSpCategoryMap size = u ∧
              categoriesAssigned = 0
        >                                                    "DL1-02"
        sequential
        [] maximumConnections := 8
                    "Maximum number of simultaneous user connections"
        [] maximumServiceCategories := 1
                    "Maximum number of service categories"
        [] maximumServiceProviders       := 3
                    "Maximum number of service providers"
        [] maximumAllowableTimeout := 5
                    "Maximum allowable timeout"
        [] srCategoryNames := SmalltalkLibPkg:::OrderedCollection
              new: maximumServiceCategories
        [] srCategoryNames addLast: 'Default Service Request category'
        [] spCategoryNames := SmalltalkLibPkg:::OrderedCollection
              new: maximumServiceProviders
        "Multiple service providers may belong to the same service
        provider category, but it is also possible that each service
        provider could belong to a different service provider category.
        The maximum size is therefore used when spCategoryNames is
        created."
        [] spCategoryNames addLast: 'Default Service Provider category'
        [] srToSpCategoryMap = Dictionary new
        [] srToSpCategoryMap at: 'Default Service Request category'
              put: spCategoryNames
        [] categoriesAssigned := 0
        end-sequential


category accessing

        message pattern maximumConnections
        method properties
        "Total correctness"
        true results-in methodReturnValue = maximumConnections "DL1-03"
        sequential
        ^maximumConnections
        end-sequential


        message pattern maximumServiceCategories
        method properties
        "Total correctness"
        true results-in methodReturnValue = maximumServiceCategories
                                                             "DL1-04"
        sequential
        ^maximumServiceCategories
        end-sequential
```

```
        message pattern maximumServiceProviders
        method properties
        "Total correctness"
        true results-in methodReturnValue = maximumServiceProviders
                                                                "DL1-05"
        sequential
        ^maximumServiceProviders
        end-sequential


        message pattern maximumAllowableTimeout
        method properties
        "Total correctness"
        true results-in methodReturnValue = maximumAllowableTimeout
                                                                "DL1-06"
        sequential
        ^maximumAllowableTimeout
        end-sequential


        message pattern getSPCategories: srCategory
        method properties
        "The srCategory passed as a parameter is used as the index into
        the srToSpCategoryMap object in order to extract the collection
        of Service Provider Categories associated with the srCategory."
        "Total correctness"
        true results-in
              methodReturnValue = srToSpCategoryMap at: srCategory
                                                                "DL1-07"
        sequential
        ^srToSpCategoryMap at: srCategory
        end-sequential


category modifying

        message pattern assignSRCategory
        method properties
        "Returns a unique service request category (each service request
        category is only assigned once)"
        "Total correctness"
        <∀ x where 0 ≤ x < maximumServiceCategories ::
        categoriesAssigned = x results-in
              categoriesAssigned = x + 1 ∧
              methodReturnValue = srCategoryNames at: (x + 1)
        >                                                       "DL1-08"
        sequential
        categoriesAssigned := categoriesAssigned + 1
        [] ^srCategoryNames at: categoriesAssigned
        end-sequential


        message pattern assignSPCategory
        method properties
        "Returns a service provider category (each service provider
        category may be assigned multiple times).  Subclasses may use
        various algorithms to assign service provider categories"
```

```
"Total correctness"
<∀ x where spCategoryNames includes: x ::
true results-in
      methodReturnValue = x
>                                                    "DL1-09"

sequential
^spCategoryNames first
end-sequential
```

B.5 The EventSimulator class

The EventSimulator class is an abstract class. It is responsible for starting a timer if one is required. It also detects the expiry of the timer. The subclasses of EventSimulator are responsible for determining **when** a timer is required and also for **generating the events** resulting from the expiry of the timers.

class EventSimulator
superclass Object **from** SmalltalkLibRepository
instance variable names
rand
"This variable refers to an instance of the Random class from the Smalltalk library. The instance is created when the EventSimulator subclass is instantiated. The instance of the Random class maintains a seed from which the next random number is generated. The random number is used to start a timer with a random value."
newEventRequired
"When the value is equal to true it means that a new event is **required.** Once the variable has been set to true, a random timer will be started at some point afterwards. When the timer is started, newEventRequired is set to false. It is the responsibility of the subclass to set this variable to true when a new event is required, since each subclass will have its own conditions for requiring a new event. Once the timer expires, an event will be generated, as will be described in the comments section of the **generatingEvent** variable."
currentRandomTimeoutValue
"This variable contains the value of the random timeout currently being requested. The purpose of this variable is to provide a mechanism for referencing the current timeout value in the correctness arguments. Note that the SLOOP statements could therefore have been rewritten without this variable while still providing the same functionality. However, in that case it would not have been possible to formalise certain correctness properties (such as DL1-04)."
generatingEvent
"The value is equal to true if the timer has expired and an event has to be **generated**, otherwise it is equal to false. The subclass sets this variable to false at the time when the event is generated. The actual event that is generated is also the reponsibility of the subclass, since each subclass will generate a different type of event."
timerOutstanding
"This variable is set to true when a timer is started and it is set to false when a timeoutElement is removed from the timerEventQ (i.e. when an expired timer has been processed). The purpose of this variable is to provide a mechanism for reasoning about the uniqueness of outstanding timers in the EventSimulator class. In this class only one timer requested by the EventSimulator may be outstanding at a time. The timerOutstanding variable is used in the preconditions of the startRandomTimer:withMaximum: method as well as in the postconditions of the resetTimerExpired: method. If subclasses need to support multiple simultaneous timers, then the preconditions of the startRandomTimer:withMaximum: method need to be weakened and the postconditions of the resetTimerExpired: method need to be strengthened. Since the purpose of the timerOutstanding variable is to facilitate correctness reasoning, the SLOOP statements could have been rewritten without this variable while still providing the same functionality."

```
timerId
      "This variable contains the identifier of the timer currently
      being requested.

class properties

      "Liveness"
      "When a simulation event is required, a simulation timer is eventually started."
                                                                  "AL2-01"

      "Liveness"
      "If a simulator timer expires, the simulator eventually has to generate an event."
                                                                  "AL2-02"

      "Clean behaviour"
      <∀ anObject ::
                  invariant anObject class ~~ EventSimulator
      >                                                           "DS2-01"
      "The EventSimulator class is an abstract class and should not be instantiated"

      "Clean behaviour"
      invariant    rand notNil  ∧ rand class = Random             "DS2-02"
      "Once rand has been initialized to refer to an instance of the Random class, it is never
      set to nil while the instance of the EventSimulator subclass exists."
      "It is therefore possible for the EventSimulator subclass instance to send messages
      to rand at any stage after initialization."

      "Clean behaviour"
      "The currentRandomTimeout value is always within the range specified by the
      precondition of the start:id:for: method of the TimerServices class."
                                                                  "DS2-03"

      "Global invariant"
      "All outstanding timers requested by an EventSimulator subclass instance are
      identified uniquely with respect to the requestor."
                                                                  "DS3-01"
      "Thus, all the timers requested by this requestor that are
      currently running or that are in the timerEventQ are uniquely
      identified with respect to the requestor."


instance methods
category private
      message pattern initialize
      "Creates an instance of class Random and sets newEventRequired,
      generatingEvent and timerOutstanding to false.  It also sets
      currentRandomTimeoutValue to 1 so that the class invariant
      referring to it will hold after instance creation and
      initialization have been completed."
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧
      rand notNil ∧ newEventRequired = false ∧
      currentRandomTimeoutValue = 1 ∧
      generatingEvent = false ∧
      timerOutstanding = false                                    "DL1-01"

      sequential
      rand := SmalltalkLibPkg:::Random new
      [] newEventRequired := false
      [] currentRandomTimeoutValue := 1
```

```
        [] generatingEvent := false
        [] timerOutstanding := false
        end-sequential

category accessing
        message pattern nextRandomNumber: maximumValue
        method properties
        "Returns the next random number between 1 and maximumValue
        inclusive"
        "Total correctness"
        true results-in
                methodReturnValue ≥ 1 ∧ methodReturnValue ≤ maximumValue
                                                                "DL1-02"
        sequential
        ^ (rand next * maximumValue) truncated + 1
        end-sequential

category testing
        message pattern timerExpired: timerEventQ
        method properties
        "Returns true if the timerEventQ contains an element of which
        the requestor == self, otherwise it returns false.  This method
        needs to be overridden when multiple simultaneous timers may be
        originated by the same requestor.  In that case the identifier
        which uniquely identifies the timer with respect to the
        requestor has to match as well."
        "Total correctness"
        true results-in methodReturnValue =
                (timerEventQ detect: [:each |
                each timeoutRequestor == self ]
                ifNone: [nil]) notNil                           "DL1-03"
        sequential
        ^ (timerEventQ detect: [:each | each timeoutRequestor == self]
                ifNone: [nil]) notNil
        end-sequential


category modifying
        message pattern startRandomTimer: aTimerServices
                        withMaximum: maximumValue
        "Start a timer with a random value within the range between 1
        and maximumValue. When the resulting start:id:for: message is
        sent to the TimerServices instance, a reference to the requestor
        (in this case the EventSimulator subclass instance) as well as
        an identifier are passed as parameters.  The combination of the
        reference to the requestor and the identifier ensures that each
        timer request can be identified uniquely within the system.
        This facilitates the correlation of the subsequent timeout
        notifications with the timer requests.

        In the EventSimulator class only one timer is outstanding at a
        time for a specific requestor, i.e. ¬timerOutstanding is a
        precondition for starting a new timer for a specific instance of
        an EventSimulator subclass. Since the timers initiated by a
        specific EventSimulator subclass instance do not run
        concurrently, these timers can all have an identifier of 1.

        If a subclass requires multiple concurrent timers, unique values
        must be allocated to the corresponding identifiers. The
        startRandomTimer:withMaximum:  method therefore needs to be
        overridden in order to achieve this. The total correctness
        property of the modified method also needs to be updated, viz. a
        disjunction needs to be added to the precondition to state that
        the proposed identifier of any new timer requested by that
```

EventSimulator subclass instance should not match any identifier of any other outstanding timer requested by that EventSimulator subclass instance. Thus, the precondition has to be **weakened**. In that case the value of timerOutstanding will no longer be relevant."

method properties
"Total correctness"
¬timerOutstanding **results-in** methodReturnValue = self ∧
self postconditions: (#nextRandomNumber:)
withArguments: #(maximumValue) ∧
aTimerServices postconditions: (#start:id:for:)
withArguments: #(self timerId currentRandomTimeoutValue) ∧
timerOutstanding **"DL1-04"**
sequential
currentRandomTimeoutValue :=
(self nextRandomNumber: maximumValue)
[] timerId := 1
[] aTimerServices start: self id: timerId for:
currentRandomTimeoutValue
[] timerOutstanding := true
end-sequential

message pattern resetTimerExpired: timerEventQ
method properties
"Removes the first timeoutElement in timerEventQ where the requestor matches the receiver."
"Total correctness"
<∃ expiredTimeout **where** expiredTimeout ==
(timerEventQ detect: [:each | each timeoutRequestor == self]
ifNone: [nil]) ::
expiredTimeout notNil **results-in**
methodReturnValue = self ∧
(timerEventQ includes: expiredTimeout) not ∧
timerOutstanding not
> **"DL1-05"**
sequential
timerEventQ remove:
(timerEventQ detect: [:each | each timeoutRequestor == self])
[] timerOutstanding := false
end-sequential

category cyclic
message pattern p_simulate: aTimerServices timeoutEventsIn: timerEventQ
"If a new event is required, start a random timer, the expiry of which will cause an event to be initiated."

method properties
"This method ensures that properties **DS2-03, AL2-01** and **AL2-02** are satisfied by the EventSimulator class."

"Clean behaviour"
invariant currentRandomTimeoutValue > 0 ∧
currentRandomTimeoutValue ≤ aTimerServices maximumTimeout **"DS2-03"**
"A timeout requested by the EventSimulator subclass instance is always within the range that ensures that the precondition of the start:id:for: method of the

TimerServices class is met when the EventSimulator subclass instance invokes that method."

```
"Precedence"
newEventRequired ensures
      self postconditions: (#startRandomTimer:withMaximum:)
      withArguments:
      #(aTimerServices (aTimerServices maximumTimeout))
      ∧ ¬newEventRequired                                    "DP1-01"
```

"When newEventRequired is true, it ensures that a simulation timer is started and newEventRequired becomes false."

```
"Precedence"
self timerExpired: timerEventQ ensures
      generatingEvent ∧
      self postconditions: (#resetTimerExpired:)
      withArguments: #(timerEventQ)                          "DP1-02"
```

"When a simulation timer expires, it ensures that generatingEvent becomes true."

```
parallel
self startRandomTimer: aTimerServices withMaximum:
(aTimerServices maximumTimeout) \+
newEventRequired := false
      if newEventRequired
[] generatingEvent := true \+
self resetTimerExpired: timerEventQ
      if self timerExpired: timerEventQ
end-parallel
```

B.6 The CommsProviderSimulator class

The CommsProviderSimulator class simulates the actions of the communication provider. This class is replaced by the CommunicationProviderAgent class when the call centre interacts with the actual communication provider instead of simulating its actions.

The CommsProviderSimulator class is a subclass of EventSimulator. The newEventRequired instance variable is set to true as part of the initialization procedures. That results in the starting of a new timer. Once the timer has expired, an event is generated. A new connection is established if one is available and the associated service request is added to the end of the input queue. The timeout is ignored if all the connections are busy. This would correspond to a busy signal being received by the service user in an actual implementation.

```
class CommsProviderSimulator
superclass EventSimulator from ApplicationsRepository
class properties
"The communication provider agent constructs the serviceRequest
associated with the connection based on information received from the
communication provider. For example, in the one case the service user
identification information and the type of service required will be
received. In another case no information will be received (perhaps
because it is not relevant, e.g. when a directory enquiry is made).
The simulation puts no information into the service request."

class methods
category instance creation
      message pattern startSimulation
      method properties

      "The initialize method of the superclass sets newEventRequired
      to false. This method sends the initialize message to its
      superclass and immediately sets newEventRequired to true. The
      values of the other instance variables mentioned in the total
      correctness property of the initialize method of the superclass
      remain unchanged."
      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      ^(super new initialize) moreInit
      end-sequential

instance methods
category private
      message pattern moreInit
      "Sets newEventRequired to true (which was set to false in the
      initialization routine of the superclass)."
      method properties
      "Total correctness"
      true results-in methodReturnValue = self  ∧
            newEventRequired = true                        "DL1-02"
      "This property ensures that property AS3-01, which was
      identified during the analysis phase, is achieved. Property
      AS3-01 specifies that instance creation results in a
      communication provider simulator event being required."
```

```
        sequential
        newEventRequired := true
        end-sequential

category accessing
        message pattern getIdleConnection: userConnections
        "Returns the first connection that is idle"
        method properties
        "Total correctness"
        true results-in methodReturnValue =
                userConnections detect: [:each | each isIdle]
                ifNone: [nil]                                          "DL1-03"
        sequential
        ^ userConnections detect: [:each | each isIdle] ifNone: [nil]
        end-sequential

category modifying
        message pattern terminate: aConnection cause: reason
        method properties
        "Inform the communication provider that the connection has
        terminated."
        "Total correctness"
        true results-in methodReturnValue = self                       "DL1-04"
        sequential
        Transcript show: 'Connection'
        [] Transcript show: (aConnection connectionIndex printString)
        [] Transcript show: 'has terminated with cause'
        [] Transcript show: reason
        end-sequential


category cyclic
        message pattern p_generateEvent: userConnections
                target: inputQ
        method macros
        idleConnection ≡ self getIdleConnection: userConnections
        method properties
        "Simulate an event from the communication provider."

        "Safe liveness"
        generatingEvent ∧ idleConnection notNil ∧
        ¬(inputQ includes:(idleConnection serviceRequest)) ensures
                ¬generatingEvent ∧ newEventRequired ∧
                inputQ last = (idleConnection serviceRequest) ∧
                idleConnection postconditions: (#assign)               "AP1-01"
        "If an event has to be generated and the maximum number of connections have not yet
        been established, the communication provider simulator ensures that a new connection
        is established, the associated service request is appended to the input queue and a new
        communication provider simulator event is again required."

        "Safe liveness"
        generatingEvent ∧ idleConnection isNil ensures
                ¬generatingEvent ∧ newEventRequired                    "AP1-02"
        "If an event has to be generated and the maximum number of connections have already
        been established, the communication provider simulator ensures that the event is
        cancelled and a new communication provider simulator event is again required."

        parallel
        inputQ addLast: (idleConnection serviceRequest) \+
        idleConnection assign
                if generatingEvent and: [idleConnection notNil] ~
```

```
Transcript show: 'All connections busy'
      if generatingEvent and: [idleConnection isNil]
|| newEventRequired := true \+
generatingEvent := false
      if generatingEvent
end-parallel
```

B.7 The Connection class

```
class Connection
superclass Object from SmalltalkLibRepository
instance variable names
state
      "The state of the connection"
terminatingReason
      "The reason why the connection is being terminated."
serviceRequest
      "The service request associated with the connection."
currentHandlerInformed
      "This flag is used when a connection has to be terminated.  It
      indicates whether the communication provider has been informed
      of the termination of the connection."
connectionIndex
      "The index of this connection into the userConnections array."


class properties
"Note that there are no safety properties specifying the allowed
values of the state instance variable.  There are also no safety
properties specifying the allowed state transitions.  The reason for
this is to avoid overspecification, i.e. it avoids restricting
subclasses to certain specified values.  Recall that preconditions may
not be strengthened and postconditions may not be weakened during
subclassing."


class methods
category instance creation
      message pattern setup: indexOfConnection
      method properties
      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      super new initialize: indexOfConnection
      end-sequential


instance methods
category private
      message pattern initialize: indexOfConnection
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧ state = 'IDLE' ∧
      serviceRequest notNil ∧ currentHandlerInformed = false ∧
      connectionIndex = indexOfConnection                  "DL1-11"
      sequential
      state := 'IDLE'
      [] serviceRequest := CC_CorePkg:::ServiceRequest setup: self
      [] currentHandlerInformed := false
      [] connectionIndex := indexOfConnection
      end-sequential
```

```
category accessing
      message pattern terminatingReason
      method properties
      "Total correctness"
      true results-in methodReturnValue = terminatingReason   "DL1-02"
      sequential
      ^ terminatingReason
      end-sequential


      message pattern connectionIndex
      method properties
      "Total correctness"
      true results-in methodReturnValue = connectionIndex     "DL1-03"
      sequential
      ^ connectionIndex
      end-sequential


      message pattern serviceRequest
      method properties
      "Total correctness"
      true results-in methodReturnValue = serviceRequest      "DL1-04"
      sequential
      ^ serviceRequest
      end-sequential


category testing
      message pattern isIdle
      method properties
      "Total correctness"
      true results-in methodReturnValue = (state = 'IDLE')    "DL1-05"
      sequential
      ^ state = 'IDLE'
      end-sequential


      message pattern isTerminating
      method properties
      "Total correctness"
      true results-in methodReturnValue = (state = 'TERMINATING')
                                                              "DL1-06"
      sequential
      ^ state = 'TERMINATING'
      end-sequential


category modifying
      message pattern assign
      method properties
      "Total correctness"
      state = 'IDLE' results-in methodReturnValue = self ∧ state =
      'CONNECTED'                                             "DL1-07"
      sequential
      state := 'CONNECTED'
            if state = 'IDLE'
      end-sequential
```

```
message pattern terminate: reason
method properties
"Total correctness"
state = 'CONNECTED' results-in
      methodReturnValue = self ∧
      state = 'TERMINATING' ∧ terminatingReason = reason
                                                          "DL1-08"
"Total correctness"
state = 'TERMINATING' results-in methodReturnValue = self
                                                          "DL1-09"
"This allows for terminate collision."

"Total correctness"
state = 'IDLE' results-in methodReturnValue = self        "DL1-10"
"This ensures that the transition from 'IDLE' to 'TERMINATING'
is not possible"
sequential
terminatingReason := reason \+
state := 'TERMINATING'
      if state = 'CONNECTED'
end-sequential


category cyclic
      message pattern p_informCommsProvider: commsAgent
      method properties
      "Safe liveness"
      state = 'TERMINATING' ∧ terminatingReason = 'completed' ∧
      ¬currentHandlerInformed ensures
            commsAgent postconditions: (#terminate:cause:)
            withArguments: #(self terminatingReason) ∧
            currentHandlerInformed                        "DP1-01"
      parallel
      commsAgent terminate: self cause: terminatingReason \+
      currentHandlerInformed := true
            if state = 'TERMINATING' and:
            [(terminatingReason = 'completed')
            and: [currentHandlerInformed not]]
      end-parallel


      message pattern p_doWrapUp
      method properties
      "Safe liveness"
      currentHandlerInformed ensures
      state = 'IDLE' ∧ serviceRequest postconditions: (#reset) ∧
      ¬currentHandlerInformed                             "DP1-02"
      parallel
      state := 'IDLE' \+
      serviceRequest reset \+
      currentHandlerInformed := false
            if currentHandlerInformed
      end-parallel
```

B.8 The ServiceCategoryAllocator class

```
class ServiceCategoryAllocator
superclass Object from SmalltalkLibRepository
instance variable categorising
      "The categorising variable is used as a flag to indicate whether
      the categorisation of the service request at the head of the
      inputQ has been initiated or not."
class properties
"Safe liveness"
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ first = aServiceRequest ∧ ¬categorising ensures
      inputQ first = aServiceRequest ∧ categorising
>                                                          "DP1-01"
"Safe liveness"
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ first = aServiceRequest ∧ categorising ensures
      < ∃ aServiceQueue where
      (scContainer detects: [:each | each serviceQ = aServiceQueue]
      ifNone: [nil]) notNil ::
      aServiceQueue includes: aServiceRequest)
      > ∧ ¬(inputQ includes: aServiceRequest) ∧ ¬(categorising)
>                                                          "DP1-02"

class methods
category instance creation
      message pattern setup
      method properties
      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      super new initialize
      end-sequential

instance methods
category private
      message pattern initialize
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧ ¬categorising
                                                           "DL1-02"
      sequential
      categorising := false
      end-sequential

category modifying
      message pattern categoriseServiceRequest: serviceRequest
            using: scContainer
      "When the service request does not contain any categorisation
      data, it is categorised as belonging to the first service
      category in scContainer.  This is the default behaviour which
      facilitates usage of this class without further subclassing if
      only one service category is supported by the system.  This
      method needs to be reimplemented in the subclasses if multiple
      service categories are supported.  In that case the default
      category is only used if the categorisation data is not
```

```
        provided, otherwise the service request is categorised according
        to the data provided by the service user (e.g. via the IVR)."
        method properties
        "Total correctness"
        true results-in
              methodReturnValue = self ∧
              serviceRequest serviceRequestCategory notNil ∧
              (scContainer detects:
              ([:each | each serviceQCategory =
              serviceRequest serviceRequestCategory] ifNone: [nil]))
              notNil                                              "DL1-03"
        sequential
        serviceRequest serviceRequestCategory:
        (scContainer first) serviceQCategory
        end-sequential

        message pattern assignToSQ: serviceRequest using: scContainer
        method macros
        match ≡ scContainer detect: [:each | each serviceQCategory =
              serviceRequest serviceRequestCategory] ifNone: [nil]
        method properties
        "Further refinements would override this method to include error
        conditions.  For example, the service request would be rejected
        if the service providers in the service provider container were
        all operating in the restricted mode."

        "Total correctness"
        serviceRequest serviceQ isNil ∧
        serviceRequest serviceRequestCategory notNil ∧ match notNil
        results-in
              methodReturnValue = self ∧
              serviceRequest serviceQ notNil ∧
              serviceRequest serviceRequestCategory notNil ∧
              (match serviceQ) includes: serviceRequest           "DL1-04"
        sequential
        serviceRequest serviceQ: (match serviceQ) \+
        (match serviceQ) addLast: serviceRequest
              if match notNil
        end-sequential


category cyclic
        message pattern p_categorise: inputQ using: scContainer
        method properties
        "Safe liveness"
        ¬(inputQ isEmpty) ∧ ¬categorising until
              categorising ∧
              self postconditions: (#categoriseServiceRequest:using:)
              withArguments: #((inputQ first) scContainer)        "DP1-03"
        parallel
        categorising := true \+
        self categoriseServiceRequest: (inputQ first)
        using: scContainer
              if inputQ isEmpty not and: [categorising not]
        end-parallel

        message pattern p_allocate: scContainer from: inputQ
        method macros
        serviceRequest ≡ inputQ first
                          if inputQ isEmpty not ~
                        nil
                          if inputQ isEmpty
```

```
method properties
"Safe liveness"
<∀ aServiceRequest where
¬(inputQ isEmpty) ∧ inputQ first == aServiceRequest ::
      aServiceRequest serviceRequestCategory notNil ∧
      aServiceRequest serviceQ isNil ∧
      < ∃ aServiceCategory where
      scContainer includes: aServiceCategory ::
            aServiceCategory serviceQCategory =
            aServiceRequest serviceRequestCategory
      >
ensures
      self postconditions: (#assignToSQ:using:)
      withArguments: #(aServiceRequest scContainer) ∧
      ¬categorising ∧
      ¬(inputQ includes: aServiceRequest)
>                                                          "DP1-04"
parallel
self assignToSQ: serviceRequest using: scContainer \+
categorising := false \+
inputQ removeFirst
      if serviceRequest notNil and:
      [serviceRequest serviceRequestCategory notNil and:
      [serviceRequest serviceQ isNil]]
end-parallel
```

B.9 The ServiceRequest class

```
class ServiceRequest
superclass Object from SmalltalkLibRepository
instance variable names
serviceQ
      "The service queue to which the service request has been
      assigned."
serviceRequestCategory
      "The category of this service request."
connection
      "The connection associated with this service request."
serviceProvider
      "The service provider to which this service request has been
      assigned."
categorisationData
      "The data which is used to categorise this service request."

class properties

class methods
category Instance creation
      message pattern setup: aConnection
      method properties
      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      ^super new initialize: aConnection
      end-sequential


instance methods
category private
      message pattern initialize: aConnection
      method properties
      true results-in methodReturnValue = self ∧
            connection = aConnection ∧
            self postconditions: (#reset)                  "DL1-02"
      sequential
      connection := aConnection
      [] self reset
      end-sequential


category accessing
      message pattern serviceQ
      method properties
      "Total correctness"
      true results-in methodReturnValue = serviceQ         "DL1-03"
      sequential
      ^ serviceQ
      end-sequential
```

```
        message pattern serviceRequestCategory
        method properties
        "Total correctness"
        true results-in methodReturnValue = serviceRequestCategory
                                                                    "DL1-04"
        sequential
        ^ serviceRequestCategory
        end-sequential

        message pattern connection
        method properties
        "Total correctness"
        true results-in methodReturnValue = connection          "DL1-05"
        sequential
        ^ connection
        end-sequential

        message pattern serviceProvider
        method properties
        "Total correctness"
        true results-in methodReturnValue = serviceProvider     "DL1-06"
        sequential
        ^ serviceProvider
        end-sequential

        message pattern categorisationData
        method properties
        "Total correctness"
        true results-in methodReturnValue = categorisationData "DL1-07"
        sequential
        ^ categorisationData
        end-sequential


category modifying
        message pattern serviceQ: sq
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧ serviceQ = sq
                                                                    "DL1-08"
        sequential
        serviceQ := sq
        end-sequential


        message pattern serviceRequestCategory: srCategory
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧
              serviceRequestCategory = srCategory               "DL1-09"
        sequential
        serviceRequestCategory := srCategory
        end-sequential

        message pattern serviceProvider: sp
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧
              serviceProvider = sp                              "DL1-10"
        sequential
        serviceProvider := sp
        end-sequential
```

```
message pattern categorisationData: newData
method properties
"Total correctness"
true results-in methodReturnValue = self ∧
      categorisationData = newData                          "DL1-11"
sequential
categorisationData := newData
end-sequential

message pattern reset
method properties
"Total correctness"
true results-in methodReturnValue = self ∧ serviceQ isNil ∧
      serviceRequestCategory isNil ∧ serviceProvider isNil ∧
      categorisationData isNil                              "DL1-12"
sequential
serviceQ := nil
[] serviceRequestCategory := nil
[] serviceProvider := nil
[] categorisationData := nil
end-sequential
```

B.10 The ServiceCategory class

Subclasses may override the *canAssignSR* and *assignToSP* methods to implement algorithms to assign service requests to service providers as required by individual applications. The algorithm implemented for the ServiceQueueCategory class is to assign the service request to the first available service provider in the container.

In anticipation of a refinement which might require that service providers be allocated in a round robin fashion, OrderedCollection rather than Set is used for the service provider container component of the ServiceCategory class. The main difference between OrderedCollection and Set is the fact that the elements of the former are ordered and those of the latter are not. By ordering the service providers within such a container, it makes it possible to implement an algorithm which will ensure that all service providers are utilised. For example, if service requests of a specific category arrive slowly enough that the second service request is only considered once the service provider has finished serving the first, then it could happen that the service requests are always allocated to the same service provider. An OrderedCollection implementation could assist in ensuring that the service requests are allocated to the service providers in a round robin fashion.

```
class ServiceCategory
superclass Object

instance variable names
serviceQCategory
      "The category of the service requests enqueued in the serviceQ"
serviceQ
      "The FIFO queue containing service requests matching
      serviceQCategory"
spCategories
      "The collection of service provider categories that apply to the
      serviceQ"
spSubset
      "The collection of service providers that may service the
      serviceQ"

class properties

class methods
category instance creation
      message pattern setup: config
      method properties
      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k ∧ config notNil results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      ^super new initialize: config
      end-sequential
instance methods
category private
      message pattern initialize: config
      method macros
      maxConn ≡ config maximumConnections
```

```
maxSP ≡ config maximumServiceProviders

method properties
"Total correctness"
config notNil results-in
      methodReturnValue = self ∧
      serviceQ notNil ∧ serviceQCategory notNil ∧
      spSubset notNil ∧ spCategories notNil ∧
      self postconditions: (#registerSPCategories:)
      withArguments: #(config)                    "DL1-02"
sequential
serviceQ := SmalltalkLibPkg:::OrderedCollection new: maxConn
[] serviceQCategory := config assignSRCategory
[] spSubset :=
      SmalltalkLibPkg:::OrderedCollection new: maxSP
[] spCategories := SmalltalkLibPkg:::OrderedCollection new
[] self registerSPCategories: config
end-sequential

message pattern registerSPCategories: config
method properties
"Total correctness"
config notNil ∧ serviceQCategory notNil results-in
      methodReturnValue = self ∧ ¬spCategories isEmpty
                                                  "DL1-03"
sequential
spCategories addAll:
      (config getSPCategories: serviceQCategory)
end-sequential


category testing
      message pattern servicedBy: spCategory
      method properties
      "Total correctness"
      spCategory notNil results-in
            methodReturnValue = (spCategories includes: spCategory)
                                                  "DL1-04"
      sequential
      ^spCategories includes: spCategory
      end-sequential


      message pattern canAssignSR
      method properties
      "Total correctness"
      true results-in methodReturnValue =
            (spSubset detect:
            [:each | each canAcceptNextSR: serviceQCategory]
            ifNone: [nil] ) notNil                "DL1-05"
      sequential
      ^ ( (spSubset detect:
            [:each | each canAcceptNextSR: serviceQCategory]
            ifNone: [nil]) notNil)
      end-sequential


category accessing
      message pattern spSubset
      method properties
      "Total correctness"
      true results-in methodReturnValue = spSubset    "DL1-06"
```

```
        sequential
        ^spSubset
        end-sequential


        message pattern serviceQCategory
        method properties
        "Total correctness"
        true results-in methodReturnValue = serviceQCategory    "DL1-07"
        sequential
        ^serviceQCategory
        end-sequential


        message pattern serviceQ
        method properties
        "Total correctness"
        true results-in methodReturnValue = serviceQ            "DL1-08"
        sequential
        ^serviceQ
        end-sequential


category modifying
        message pattern addSP: anSP
        method properties
        "Total correctness"
        anSP notNil results-in
                methodReturnValue = self ∧ spSubset includes: anSP
                                                                "DL1-09"
        sequential
        spSubset addLast: anSP
        end-sequential

        message pattern assignToSP: sr
        method macros
        availableServiceProvider ≡
                spSubset detect: [:each | each canAcceptNextSR:
                serviceQCategory]
        method properties
        "Total correctness"
        sr notNil ∧ availableServiceProvider notNil results-in
        methodReturnValue = self ∧
        sr postconditions: (#serviceProvider:)
                withArguments: #(availableServiceProvider) ∧
        availableServiceProvider
                postconditions:(#processServiceRequest:)
                withArguments: #(sr)                            "DL1-10"
        sequential
          sr serviceProvider: availableServiceProvider
        [] availableServiceProvider processServiceRequest: sr
        end-sequential


category cyclic
        message pattern p_execute
        method properties
        "Safe liveness"
        serviceQ isEmpty not ∧ self canAssignSR ensures
                self postconditions: (#assignToSP:) withArguments:
                #((serviceQ first)) ∧
                serviceQ postconditions: (#removeFirst)         "DP1-01"
```

```
parallel
self assignToSP: (serviceQ first) \+
serviceQ removeFirst
      if serviceQ isEmpty not and: [self canAssignSR]
end-parallel
```

B.11 The TimerServices class

The TimerServices class allows its clients to request that timers of specified durations be started. The timer resolution is in seconds and the maximum timeout value is denoted by maximumTimeout. The TimerServices class uses a **circular** array called timeoutCollection to implement the timers. Each position in the array represents one second. The object maintains an index into the array. This index is called currentTick and is advanced every second. Thus, the entry in the array which will be reached *x* seconds from the current moment can be calculated using the value of currentTick, the size of the array and the value of *x*.

Each entry in the array is an ordered collection of TimeoutElement instances. When the TimerServices class receives a request to start a timer, it creates a TimeoutElement instance to represent that timer and enters it into the relevant collection of TimeoutElement instances, i.e. it enters it into the collection that will be reached after *x* seconds, where *x* is the timeout value specified for the timer. The TimeoutElement instance contains the timeout identifier. It also contains other attributes that may be used to obtain information about the timeout.

When a timer expires, its associated TimeoutElement instance is removed from the relevant collection in the circular array and it is added to timerEventQ. The latter is inspected by the clients of TimerServices in order to determine whether their requested timers have timed out yet.

The currentTick variable is used to calculate the read and write indices and is updated every second. When the TimerServices instance is created, currentTick is initialized to zero. Every second it is incremented modulo the size of the array. The size of the array is a function of the maximum timeout value.

Thus, when a TimerServices instance receives a request to start a timer, it has to determine which list of TimeoutElement instances will be processed after (duration + 1) clock ticks. It is necessary to add the extra one in order to ensure that a timer does not expire prematurely. For example, if a timer for one second is started at the time when half a second of the currentTick period has already passed, then the new timer will expire after only half a second. By adding the extra one, a timer of one second could take up to two seconds to expire, but it will at least run for one second.

The write index is therefore calculated as follows:

```
writeIndex :=
((currentTick + duration + 1) \\ (timeoutCollection size)) + 1
```

The requested timeout value (represented by duration) plus one is added to the value of currentTick. The addition is performed modulo the size of the array, which is maximumTimeout + 2. The size of the array is explained as follows: Since one is always added to the duration in order to ensure that a timer does not expire prematurely, the maximum timeout value is actually maximumTimeout + 1. Another one has to be added to ensure that the read and write indices never have the same value. After the modulo addition has been performed, the writeIndex has to be adjusted by one, since SLOOP array indices start at one, not zero. The readIndex has the value currentTick plus one. It is also adjusted by one in order to compensate for the fact that the array indices start at one. Once the writeIndex has been calculated, the TimeoutElement instance is added to the end of the First In First Out list that is stored at that position in the array.

Figure B-3(a) illustrates where a new entry would be added for the following values of the various variables:
maximumTimeout = 5
duration = 5
currentTick = 3

Figure B-3(b) shows the result if the currentTick value had been 0 at the time of the timer request.

Note that the currentTick value ranges from 0 to 6 inclusive in the above example, whereas readIndex and writeIndex may range from 1 to 7 inclusive.

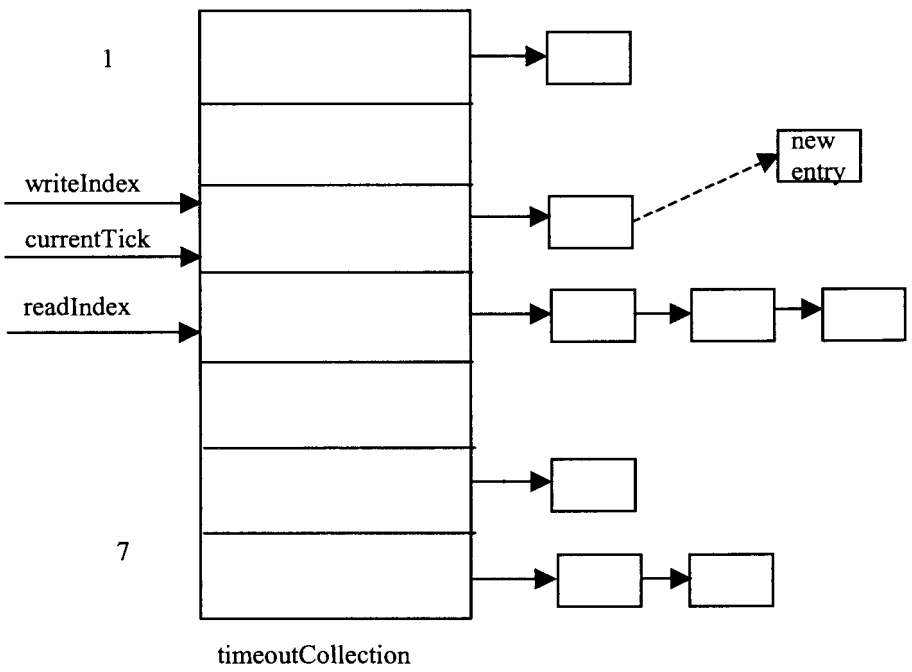

Figure B-3(a). Structures used by aTimerServices (currentTick = 3).

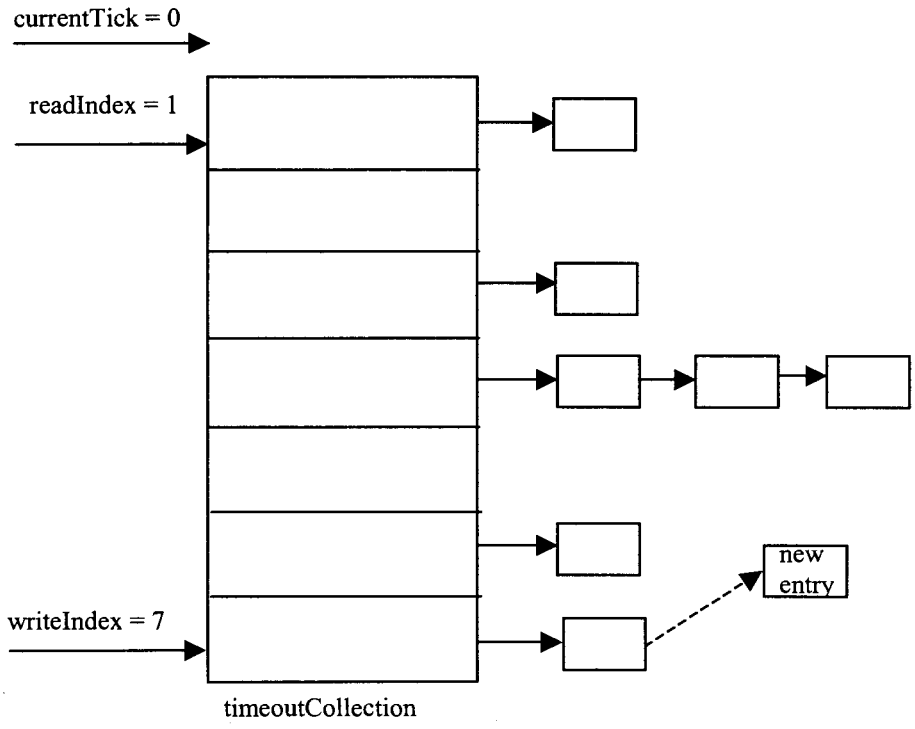

Figure B-3(b). Structures used by aTimerServices (currentTick = 0).

The value of the currentTick variable is updated whenever one second has passed since its last update and all the entries at the current readIndex have been processed. This is reflected by one of the parallel statements of the TimerServices class (it is executed infinitely often):

```
lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 and: [currentTimeoutElement isNil]
```

This statement uses two *macro-variables*, viz. difference and currentTimeoutElement. They receive their values in the *macros-section* of the method containing the parallel statement, as shown below (the *macro-variable* readIndex is used in the definition of currentTimeoutElement and therefore has to be defined prior to its usage):

```
         readIndex := currentTick + 1
      [] difference ≡ currentTime - lastTime
                        if (currentTime - lastTime) ≥ 0 ~
                      currentTime + (86400 - lastTime)
                        if (currentTime - lastTime) < 0
```

```
[] currentTimeoutElement :=
    (timeoutCollection at: readIndex) first
       if (timeoutCollection at: readIndex) isEmpty not ~
    currentTimeoutElement := nil
       if (timeoutCollection at: readIndex) isEmpty
```

One of the other parallel statements of the TimerServices class ensures that the currentTime variable is updated on a regular basis:

```
currentTime := SmalltalkLibPkg:::Time now asSeconds
```

Evaluation of the Time now asSeconds expression yields the number of seconds since midnight. The calculation of difference takes the rollover at midnight into account in the *macros-section* shown above.

The TimerServices class is now presented in the SLOOP notation.

```
class TimerServices
superclass Object from SmalltalkLibRepository
instance variable names
maximumTimeout
      "The maximum timeout value that may be requested"
timeoutCollection
      "A circular array.  Each element comprises a list of
      TimeoutElement instances"
currentTick
      "It points to a position in the timeoutCollection array. It is
      used to calculate the read and write indices."
currentTime
      "The most recent time (in number of seconds since midnight)
      obtained from the system."
lastTime
      "The time (in number of seconds since midnight) when the
      currentTick was last updated."

class properties
      "Global invariant:  when a timeout is indicated to the
      requestor, then a period greater than or equal to the value
      specified by the requestor has expired."
      invariant
      <∀ aTimeoutElement where
            timerEventQ includes: aTimeoutElement::
            aTimeoutElement timerExpired
      >                                                    "DS3-01"


      <∀ aTimeoutElement where
            <∃ i where 1 ≤ i ≤ (timeoutCollection size) ::
            (timeoutCollection at: i) includes: aTimeoutElement
            > ::
            ¬aTimeoutElement timerServicesCompleted leads-to
                        aTimeoutElement timerServicesCompleted
      >                                        "DL2-01(TimerServices)"
      "Once a timer has been started, i.e. it is present in one of the lists associated with
      timeoutCollection, the TimerServices instance will eventually complete its
      responsibilities regarding the timer (i.e. the timer will either be stopped or the
      TimerServices instance will indicate its expiry to the requestor of the timer)."
      "This property refers to the timerServicesCompleted method
      rather than the timerExpired method in order to make provision
      for subclasses that may allow a timer to be aborted."

class methods
category Instance creation
      message pattern setup: config
      method properties
      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k ∧ config notNil results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                    "DL1-01"
      sequential
      ^super new initialize: config
      end-sequential
```

```
instance methods
category private
        message pattern initialize: config
        method properties
        "Total correctness"
        true results-in timeoutCollection notNil ∧
                maximumTimeout = config maximumAllowableTimeout ∧
                < ∀ i where 1 ≤ i ≤ timeoutCollection size ::
                (timeoutCollection at: i) notNil
                > ∧
                currentTick = 0 ∧ currentTime notNil ∧ lastTime notNil
                                                                "DL1-02"
        sequential
           maximumTimeout := config maximumAllowableTimeout
        [] timeoutCollection :=
                SmalltalkLibPkg:::Array new: (maximumTimeout + 2)
        [] < [] i where 1≤i≤timeoutCollection size ::
              timeoutCollection at: i put: (OrderedCollection new)
          >
        [] currentTick := 0
        [] currentTime := SmalltalkLibPkg:::Time now asSeconds
        [] lastTime := currentTime
        end-sequential


category accessing
        message pattern maximumTimeout
        method properties
        "Total correctness"
        true results-in methodReturnValue = maximumTimeout      "DL1-03"
        sequential
        ^maximumTimeout
        end-sequential


        message pattern isTimerRunningFor: requestor with: identifier
        method properties
        "Total correctness"
        true results-in methodReturnValue = (found notNil) ∧
                found detects
                <∀ i where (1≤i≤timeoutCollection size)::
                        <∃ aTimeoutElement where
                   (timeoutCollection at:i) includes: aTimeoutElement ::
                      aTimeoutElement timeoutRequestor = requestor ∧
                      aTimeoutElement timeoutIdentifier = identifier
                      >
                >                                               "DL1-04"
        sequential
        found := nil
        [] < [] i where (1≤i≤timeoutCollection size)::
           found :=  (timeoutCollection at: i) detect:
           [:each | each timeoutRequestor = requestor and:
              [each timeoutIdentifier = identifier]] ifNone: [nil]
              if found isNil
          >
        [] ^ found notNil
        end-sequential
```

```
category modifying
      message pattern start: requestor id: identifier for: duration
      method macros
         writeIndex ≡
            ((currentTick + duration + 1)\\(timeoutCollection size))
            + 1
         "This is because the array index starts at 1, not 0"

      method properties
      "Clean behaviour"
      invariant    1 ≤ writeIndex ∧
            writeIndex ≤ timeoutCollection size                    "DS3-02"

      invariant    writeIndex ~= readIndex                         "DS3-03"

      "Total correctness"
      0 < duration ∧ duration ≤ maximumTimeout results-in
            methodReturnValue = self ∧
            nextElement class = TimeoutElement ∧
            (timeoutCollection at: writeIndex) includes: nextElement ∧
            TimeoutElement postconditions: (#setup:id:for:)
            withArguments: #(requestor identifier duration)   "DL1-05"
      sequential
      nextElement :=
      SystemUtilitiesPkg:::TimeoutElement setup: requestor
      id: identifier for: duration
            if 0 < duration ∧ duration ≤ maximumTimeout
      [] (timeoutCollection at: writeIndex) addLast: nextElement
            if 0 < duration ∧ duration ≤ maximumTimeout
      end-sequential
```

```
category cyclic
      message pattern p_runTimer: timerEventQ
      method macros
         readIndex ≡ currentTick + 1
         "This is because the array index starts at 1, not 0"

      [] difference ≡ currentTime - lastTime
                        if (currentTime - lastTime) ≥ 0 ~
                     currentTime + (86400 - lastTime)
                        if (currentTime - lastTime) < 0
      [] currentTimeoutElement ≡
         (timeoutCollection at: readIndex) first
            if (timeoutCollection at: readIndex) isEmpty not ~
         nil
            if (timeoutCollection at: readIndex) isEmpty


      method properties
      "Intermittent assertion"
      < ∀ y where 0 ≤ y < (timeoutCollection size) ::
      currentTick = y ∧ difference ≥ 1 ∧
      (currentTimeoutElement isNil leads-to
            currentTick = (y + 1) \\ (timeoutCollection size)
      >                                                            "DL2-02"
      "Thus, all the entries at the current timeout position have to
      be processed before the entries at the next position are
      processed.  The granularity of the timer is ≥ 1 second"
```

```
"Safe liveness"
<∀aTimeoutElement where
aTimeoutElement = currentTimeoutElement::
aTimeoutElement = notNil ensures
      ¬((timeoutCollection at: readIndex) includes:
      aTimeoutElement) ∧
      timerEventQ includes: aTimeoutElement ∧
      aTimeoutElement timerServicesCompleted
>                                                    "DP1-01"

parallel
   currentTime := SmalltalkLibPkg:::Time now asSeconds
[] lastTime := currentTime \+
  currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 and: [currentTimeoutElement isNil]
[] timerEventQ addLast: currentTimeoutElement \+
   currentTimeoutElement updateEndTime \+
   currentTimeoutElement timerServicesCompleted: true \+
   (timeoutCollection at: readIndex) removeFirst
      if currentTimeoutElement notNil
end-parallel
```

Note that the macro-variable writeIndex cannot be defined in the class-macros section, because it refers to the pseudo-variable duration.

The clients of the TimerServices class are not restricted to removing only the first element of the timerEventQ. That ensures that each client will receive its timeout information even if other clients misbehave. Each client must remove the TimeoutElement instance from the timerEventQ as it processes it, otherwise it will process the same element multiple times.

The advantage of having a timerEventQ is that the interface between the TimerServices instance and its clients is very loosely coupled. Alternatively the TimerServices instance can invoke a client method when a timer expires, but in that case it is necessary to reserve the client together with the TimerServices instance when the timeout is processed.

The reason for defining timerEventQ as a peer class rather than as part of an aggregation (i.e. within TimerServices) is to allow for more parallelism. This way it is not necessary to reserve timerEventQ as well whenever a timer method is executed.

B.12 The TimeoutElement class

The TimeoutElement class is now presented using the SLOOP notation:

```
class TimeoutElement
superclass Object from SmalltalkLibRepository
instance variable names
startTime
      "The time when this timeout was started"
endTime
      "The time when this timeout expired."
timeoutRequestor
      "The requestor of this timeout."
timeoutIdentifier
      "The identifier of this timeout.  It is unique with respect to
      the requestor."
requestedDuration
      "The requested duration of this timeout."
timerServicesCompleted
      "This flag indicates whether the TimerServices instance has
      completed its tasks regarding this timeout.  This flag is used
      rather than checking the endTime and startTime in order to be
      able to cater for the case where the timeout is aborted."
class macros
      currentTime ≡ SmalltalkLibPkg:::Time totalSeconds
      "currentTime contains the total number of seconds since January
      1, 1901."

class properties
      invariant    endTime ≥ startTime                          "DS3-01"
      self getCurrentDuration = 0 unless
            self getCurrentDuration > 0                         "DS4-01"

class methods
category instance creation
      message pattern setup: requestor id: identifier for: duration
      method properties
      "Total correctness"
      <∀ k where k ≥ 0 ::
      self instanceCount = k results-in
            self instanceCount = k + 1 ∧
            methodReturnValue notNil
      >                                                         "DL1-01"
      sequential
      ^super new  initialize: requestor id: identifier for: duration
      end-sequential


instance methods
category private
      message pattern initialize: requestor id: identifier for:
      duration
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧
            timeoutRequestor = requestor ∧
            timeoutIdentifier = identifier ∧
            startTime notNil ∧ endTime notNil ∧
            requestedDuration = duration ∧ ¬timerServicesCompleted
                                                                  "DL1-02"
```

```
        sequential
        timeoutRequestor := requestor
        [] timeoutIdentifier := identifier
        [] startTime := currentTime
        [] endTime := currentTime
        [] requestedDuration := duration
        [] timerServicesCompleted := false
        end-sequential


category accessing
        message pattern timeoutRequestor
        method properties
        "Total correctness"
        true results-in methodReturnValue = timeoutRequestor    "DL1-03"
        sequential
        ^timeoutRequestor
        end-sequential

        message pattern timeoutIdentifier
        method properties
        "Total correctness"
        true results-in methodReturnValue = timeoutIdentifier   "DL1-04"
        sequential
        ^timeoutIdentifier
        end-sequential

        message pattern startTime
        method properties
        "Total correctness"
        true results-in methodReturnValue = startTime           "DL1-05"
        sequential
        ^startTime
        end-sequential

        message pattern requestedDuration
        method properties
        "Total correctness"
        true results-in methodReturnValue = requestedDuration   "DL1-06"
        sequential
        ^requestedDuration
        end-sequential

        message pattern getCurrentDuration
        method properties
        "Total correctness"
        true results-in methodReturnValue = (currentTime - startTime)
                                                                "DL1-07"
        sequential
        ^(currentTime - startTime)
        end-sequential

        message pattern getTimeoutDuration
        method properties
        "Total correctness"
        true results-in methodReturnValue = (endTime - startTime)
                                                                "DL1-08"
        sequential
        ^(endTime - startTime)
        end-sequential
```

```
category testing
      message pattern timerExpired
      method properties
      " Total correctness "
      true results-in methodReturnValue =
            (self getCurrentDuration - requestedDuration ≥ 0)
                                                              "DL1-09"
      sequential
      ^(self getCurrentDuration - requestedDuration ≥ 0)
      end-sequential


      message pattern timerServicesCompleted
      method properties
      "Total correctness"
      true results-in methodReturnValue = timerServicesCompleted
                                                              "DL1-10"
      sequential
      ^timerServicesCompleted
      end-sequential


category modifying
      message pattern updateEndTime
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧
            endTime = currentTime                             "DL1-11"
      sequential
      endTime := currentTime
      end-sequential


      message pattern timerServicesCompleted: newValue
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧
            timerServicesCompleted = newValue                 "DL1-12"
      sequential
      timerServicesCompleted := newValue
      end-sequential
```

B.13 The ServiceProviderSimulator class

class ServiceProviderSimulator
superclass EventSimulator **from** ApplicationsRepository
instance variable names
serviceRequest
"This variable refers to the service request currently being serviced by the service provider simulator. Note that the reference to the ServiceRequest instance is passed to the simulator as a parameter, i.e. the ServiceRequest instance is not created by the ServiceProviderSimulator instance and therefore does not form part of it."
serviceProviderCategory
"This variable contains the name of the service provider category to which the service provider simulator belongs."
categoriesServed
"This is an ordered collection containing the names of the service request categories serviced by this service provider. The purpose of this array is to facilitate a round robin servicing scheme of these categories. That prevents starvation of a specific service category."
nrOfCategoriesServed
"This variable contains the number of service request categories serviced by this service provider. It is used in the calculation when the categoryIndex is updated."
categoryIndex
"This variable is used as index into the categoriesServed collection. It is used to determine the next service request category to be serviced by this service provider. It is incremented modulo nrOfCategoriesServed. Its values range from 0 to nrOfCategoriesServed - 1"

class properties

<∀ categoryIndex **where** categoryIndex ≥ 0 ∧
categoryIndex ≤ nrCategoriesServed - 1 ::
invariant serviceRequest notNil ⇒ ¬self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1))
>
"AS3-01"

"A service provider simulator services a single service request at a time."
"If a service request is currently assigned to the simulator, no other service request from any of the categories being served by this simulator will be served by the latter."

serviceRequest isNil ∧ ¬newEventRequired **unless**
serviceRequest notNil ∧ newEventRequired
"AS4-01"

"When a new service request is assigned to the service provider simulator then a new service provider simulator event is required."

"Note: The parent class, viz. EventSimulator, contains a parallel method which monitors the value of newEventRequired. If it detects that newEventRequired is true, it starts a timer and sets newEventRequired to false."

serviceRequest notNil ∧ ¬newEventRequired **unless**
serviceRequest isNil ∧ ¬newEventRequired
"AS4-02"

"If a service request has been assigned to the service provider simulator and newEventRequired is false, then newEventRequired remains false while the service request is still assigned to the service provider simulator."
"This has the effect that this simulator will not start another timer before the servicing of the current service request has been completed."

```
generatingEvent ∧ serviceRequest notNil ensures
      (serviceRequest connection)
      postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil ∧
      ¬generatingEvent                                    "AP1-01"
```

"If a service provider simulator has to generate an event, it ensures that the service provider simulator terminates the connection currently associated with it and becomes available to service a new service request."

"Note: The parent class, viz. EventSimulator, contains a parallel method which sets generatingEvent to true when a timer has expired."

```
<∀ aServiceRequest where serviceRequest = aServiceRequest ::
 serviceRequest = aServiceRequest ensures
      (serviceRequest connection)
      postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil
>                                                         "AP1-02"
```

"A service request remains assigned to a service provider simulator until the latter completes the service and terminates the connection."

```
invariant categoryIndex ≥ 0 ∧
      categoryIndex < nrOfCategoriesServed
                                                          "DS2-01"
```

"The categoryIndex is always greater than or equal to zero and less than nrOfCategoriesServed."

```
invariant    categoriesServed notNil ∧
             categoriesServed class = OrderedCollection
                                                          "DS2-02"
```

"Once categoriesServed has been initialized to refer to an instance of the OrderedCollection class, it is never set to nil while the ServiceProviderSimulator instance exists."

```
<∀ categoryIndex where 0 ≤ categoryIndex ∧
categoryIndex < nrOfCategoriesServed ::
¬(self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1)))
leads-to
      self canAcceptNextSR:
      (categoriesServed at: (categoryIndex + 1))
>                                                         "DL2-01"
```

"For any service category serviced by the service provider simulator, the service provider simulator will eventually be able to service a request from that service category."

```
class methods
category instance creation
      message pattern startSimulation: scContainer using:
                  aConfiguration
      method properties
      "The initialize method of the superclass sets newEventRequired
      to false.  The startSimulation:using: method invokes the
      initialize message of its superclass and initialises the
      ServiceProviderSimulator-specific instance variables.  The
      properties of the initialize method of the superclass hold."
      "Total correctness"
      true results-in methodReturnValue notNil ∧
            super postconditions: (#initialize) ∧
            serviceRequest isNil ∧
            serviceProviderCategory notNil ∧
            categoriesServed notNil ∧
            nrOfCategoriesServed ≥ 0 ∧
            categoryIndex ≥ 0
                                                      "AL1-01"
      "Instance creation results in the initialization of the instance variables of the
      ServiceProviderSimulator class and its superclasses."
      sequential
      ^(super new initialize) moreInit: scContainer using:
      aConfiguration
      end-sequential


instance methods
category private
      message pattern moreInit: scContainer using: aConfiguration
      "Initializes the ServiceProviderSimulator instance"
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧
            serviceRequest isNil ∧
            aConfiguration postconditions: (#assignSPCategory) ∧
            serviceProviderCategory notNil ∧
            categoriesServed notNil ∧
            self postconditions: (#registerServiceProvider: using:)
                  withArguments: #(scContainer aConfiguration)
                                                      "DL1-01"
      "The initialization that is performed during instance creation results in the service
      provider simulator being available to provide service and in the service provider
      simulator being registered with each service category that has a matching service
      provider category in its service provider categories component."
      sequential
        serviceRequest := nil
      [] serviceProviderCategory := aConfiguration assignSPCategory
      [] categoriesServed := OrderedCollection new
      [] self registerServiceProvider: scContainer using:
        aConfiguration
      end-sequential

category accessing
      message pattern serviceProviderCategory
      method properties
      "Total correctness"
      true results-in methodReturnValue = serviceProviderCategory
                                                      "DL1-02"
```

```
        sequential
        ^serviceProviderCategory
        end-sequential

        message pattern serviceRequest
        method properties
        "Total correctness"
        true results-in methodReturnValue = serviceRequest      "DL1-03"
        sequential
        ^ serviceRequest
        end-sequential


category testing
        message pattern canAcceptNextSR: requestingServiceCategory
        method properties
        "Total correctness"
        true results-in
              methodReturnValue = ((requestingServiceCategory =
              categoriesServed at: (categoryIndex + 1)) ∧
              (serviceRequest isNil))                           "DL1-04"
        sequential
        ^ (serviceRequest isNil)
              if requestingServiceCategory =
              categoriesServed at: categoryIndex + 1 ~
        ^false
              if requestingServiceCategory ~=
              categoriesServed at: categoryIndex + 1
        end-sequential


category modifying
        message pattern registerServiceProvider: scContainer
                    using: aConfiguration
        "Registers the ServiceProviderSimulator with the relevant
        service categories"
        method macros
        maxCategories ≡ aConfiguration maximumServiceCategories
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧
              <∀aServiceCategory where
              scContainer includes: aServiceCategory ∧
              aServiceCategory servicedBy: serviceProviderCategory ::
              aServiceCategory postconditions: (#addSP:)
              withArguments: #(self) ∧
              categoriesServed includes:
                    (aServiceCategory serviceCategory)
              > ∧
              nrOfCategoriesServed = categoriesServed size ∧
              categoryIndex ≥ 0                                 "DL1-05"
        sequential
        nrOfCategoriesServed := 0
        [] < [] j where 1≤j≤maxCategories ::
        (scContainer at: j) addSP: self \+
        nrOfCategoriesServed := nrOfCategoriesServed + 1 \+
        categoriesServed addLast:((scContainer at: j) serviceCategory)
              if (scContainer at: j)servicedBy:serviceProviderCategory
        >
        [] categoryIndex := 0
        end-sequential
```

```
message pattern processServiceRequest: aServiceRequest
method properties
"A new simulation is required each time when a new service
request is processed."
"Total correctness"
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                                                          "DL1-06"
sequential
  newEventRequired := true
[] serviceRequest := aServiceRequest
[] categoryIndex := (categoryIndex + 1) \\ nrOfCategoriesServed
end-sequential
```

```
category cyclic
      message pattern p_generateEvent
      method properties
      "The newEventRequired attribute is not updated here.  It is only
      set to true once a new service request has been received."

      generatingEvent ∧ serviceRequest notNil ensures
            (serviceRequest connection)
            postconditions: (#terminate:)
            withArguments: #('completed') ∧
            serviceRequest isNil ∧ ¬generatingEvent
                                                          "DP1-01"
```

"If a service provider simulator has to generate an event, it ensures that the connection currently associated with the service request is terminated and that the service provider simulator becomes available to service a new service request."

```
      parallel
      (serviceRequest connection) terminate: 'completed' \+
      serviceRequest := nil \+
      generatingEvent := false
            if generatingEvent
      end-parallel


      message pattern p_updateCategoryIndex: scContainer
      method properties
            <∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
            categoryIndex = x ∧
            (scContainer detect:[:each |
            (each serviceQCategory =
            (categoriesServed at: categoryIndex))
            and: [each serviceQ isEmpty]] ifNone: [nil]) notNil
            ensures
            categoryIndex = (x + 1) \\ nrOfCategoriesServed
                                                          "DP1-02"
```

"If the serviceQ of the service category matching the categoriesServed entry at the current categoryIndex+1 is empty, the categoryIndex is incremented modulo nrOfCategoriesServed.."

```
parallel
categoryIndex := (categoryIndex + 1) \\ nrOfCategoriesServed
      if (scContainer detect:[:each |
      (each serviceQCategory =
      (categoriesServed at: (categoryIndex+1)))
      and: [each serviceQ isEmpty]] ifNone: [nil]) notNil
end-parallel
```

BIBLIOGRAPHY

AbLa89 Abadi, L. and L. Lamport. Composing specifications. REX workshop on stepwise refinement of distributed systems: models, formalisms, correctness, LNCS 430, pp. 1-41, Springer-Verlag Berlin, Heidelberg 1989.

Abri96 Abrial, J.-R. The B-book - assigning programs to meanings. Cambridge Univerity Press, 1996.

ABV00 Alexander, R.T., J.M. Bieman and J. Viega. Coping with Java programming stress. Computer, pp. 30-38, April 2000.

Back89 Back, R. –J. R. Refinement calculus, part II: parallel and reactive programs. REX workshop on stepwise refinement of distributed systems: models, formalisms, correctness, LNCS 430, pp. 67-93, Springer-Verlag Berlin, Heidelberg 1989.

BaKu89 Back, R. –J. R. and R. Kurki-Suonio. Decentralization of process nets with centralized control. Distributed Computing, No. 3, pp. 73-87, 1989.

BaSe94 Back, R. –J. R. and K. Sere. From action systems to modular systems. Symposium of Formal Methods Europe (FME) 94, lecture notes in computer science, No. 873, pp. 1-25, 1994.

BaWr89 Back, R. –J. R. and J. von Wright. Refinement calculus, part I: sequential nondeterministic programs. REX workshop on stepwise refinement of distributed systems: models, formalisms, correctness, LNCS 430, pp. 42-66, Springer-Verlag Berlin, Heidelberg 1989.

BeBe97 Benveniste, A. and G. Berry. Synchronous languages and reactive system design. Proceedings of the workshop on formal design of safety critical embedded systems, pp. 60 - 86, Munich, Germany, 16-18 April 1997.

Bekk93 Bekker, C. Relationships and reflection in the object-oriented paradigm. M.Sc. dissertation, University of Pretoria, 1993.

Bena90 Ben-Ari, M. Principles of concurrent and distributed programming. Prentice-Hall International, London, 1990.

BGL93 Bruegge, B., T. Gottschalk and B. Luo. A framework for dynamic program analyzers. Proceedings of the conference on object-oriented programming: systems, languages and applications (OOPSLA), pp. 65 - 82, 1993.

Bjor99 Bjorner, D. Where do software architectures come from? A systematic development from domains and requirements; a re-assessment of software engineering. South African Computer Journal (SACJ), No. 22, pp. 3 - 13, March 1999.

BMRSS96 Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-oriented software architecture: a system of patterns. John Wiley & Sons Ltd, England, 1996.

Butl99 Butler, M. csp2B: A practical approach to combining CSP and B. Formal methods 1999, Vol. 1, LNCS 1708, pp. 490-508, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

CES71 Coffman, E.G., M. J. Elphick and A. Shoshani. System deadlocks. Computing Surveys, Vol.3, pp. 67-78, June1971.

ChCh99 Charpentier, M. and K. M. Chandy. Towards a compositional approach to the design and verification of distributed systems. Formal methods 1999, Vol. 1, LNCS 1708, pp. 570-589, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

ChMi88 Chandy, K.M. and J. Misra. Parallel program design: a foundation. Addison-Wesley, U.S.A., 1988.

CHYLS-Web Chung, P.E., Y. Huang, S. Yajnik, D. Liang, J.C. Shih, C.-Y. Wang and Y.-M. Wang. DCOM and CORBA side by side, step by step and layer by layer. On-line Web white paper at http://www.cs.wustl.edu/~schmidt/submit/Paper.html.

Dijk76 Dijkstra, E.W. A discipline of programming. Prentice-Hall International, Englewood Cliffs, N.J, 1976.

Dijk78 Dijkstra, E.W. Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands.

EnKo98 Engelbrecht, R.L. and D. Kourie. Issues in translating Smalltalk to Java. In: Compiler construction (Ed. Kai Koskimies). Proceedings of the 7th international conference, CC'98, pp. 249 - 263, March 28 - April 4, 1998.

FGHVE96 Fang, W., S. Guyet, R. Haven, M. Vilmi and E. Eckmann. VisualAge for Smalltalk Distributed – developing distributed object applications. Prentice-Hall, International, New Jersey, 1996.

Fran92 Francez, N. Program verification. Addison-Wesley, 1992.

GeRo89 Gehani, N. and W.D. Roome. The Concurrent C programming language. Prentice-Hall, 1989.

GHJV95 Gamma, E., R. Helm, R. Johnson and J. Vlissides. Design patterns, elements of reusable object-oriented software. Addison-Wesley, 1995.

GePn89 Gerth, R. and A. Pnueli. Rooting UNITY. Association for Computing Machinery (ACM), No. 1, pp. 11-19, 1989.

GoRo89 Goldberg, A. and D. Robson. Smalltalk-80, the language. Addison-Wesley, 1989.

Grie96 Gries, D. The need for education in useful formal logic. An invitation to formal methods, Computer, pp. 29 - 30, April 1996.

GrSc99 Gries, D. and F.B. Schneider. Teaching math more effectively, through the design of calculational proofs. South African Computer Journal (SACJ), No. 22, pp. 28 - 31, March 1999.

GuHo93 Guttag, J.V. and J.J. Horning. Larch: Languages and tools for formal specification. Texts and monographs in computer science. Springer-Verlag, 1993.

HaGe97 Harel, D. and E. Gery. Executable object modeling with statecharts. IEEE Computer, Vol. 30, No. 7, pp. 31-42, July 1997.

HHG90 Helm, R., I.M. Holland and D Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. Proceedings of the conference on object-oriented programming: systems, languages and applications (OOPSLA), 1990, pp. 169 - 180, October 1990.

Hoar85 Hoare, C.A.R. Communicating sequential processes. Prentice-Hall International, London, 1985.

Hoar99 Hoare, C.A.R. Theories of programming: top-down and bottom-up and meeting in the middle. Formal methods 1999, Vol. 1, LNCS 1708, pp. 1-27, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

HoHo95 Hopkins, T. and B. Horan. Smalltalk, an introduction to application development using VisualWorks. Prentice Hall, 1995.

Holz91 Holzman, G.J. Design and validation of computer protocols. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1991.

Ince93 Ince, D.C. An introduction to discrete mathematics, formal specification and Z (2nd ed). Oxford applied mathematics and computing science series, Clarendon Press, United Kingdom, 1993.

ISO89 International Standards Organization (ISO) International Standard (IS) 8807. Information processing systems -- Open Systems Interconnection -- LOTOS -- A formal description technique based on the temporal ordering of observational behaviour, 1989.

ISO97 International Standards Organization (ISO) International Standard (IS) 9074. Information technology -- Open Systems Interconnection -- Estelle: A formal description technique based on an extended state transition model. Amendment 1, 1997.

ITU-T93 International Telecommunication Union Telecommunication standardization sector (ITU-T) Recommendation Z.100. Specification and Description Language (SDL), Helsinki, March, 1993.

JiZh96 Jia, G. and G. Zheng. Fair transition system specification: an integrated approach. ACM SIGPLAN Notices, Vol. 31, No 3, pp. 14 - 21, March 1996.

JoFo88 Johnson, R. E. and B. Foote. Designing reusable classes. Journal of Object-Oriented Programming, Vol.1, No. 2, pp. 22-35, 1988.

Jone80 Jones, C. B. Software development: a rigorous approach. Prentice-Hall International, London, 1980.

Jone86 Jones, C. B. Systematic software development using VDM. Prentice-Hall International, London, 1986.

Jone96 Jones, C. B. A rigorous approach to formal methods. An invitation to formal methods, Computer, pp. 20 - 21, April 1996.

Jone99 Jones, C. B. Scientific decisions which characterize VDM. Formal methods 1999, Vol. 1, LNCS 1708, pp. 28-47, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

Kata-Web Katara, M. A short tutorial on DisCo. On-line Web tutorial at http://www.cs.tut.fi/laitos/DisCo/tutorial/Tutorial.html.

Keen89 Keene, S.E. Object-oriented programming in Common Lisp - A programmer's guide to CLOS. Addison-Wesley, 1989.

Krög87 Kröger, F. Temporal logic of programs. Springer-Verlag, Berlin, 1987.

Kurk96 Kurki-Suonio, R. Fundamentals of object-oriented specification and modeling of collective behaviors. Object-oriented behavioral specifications, pp. 101-120, Kluwer Academic, London, 1996.

LaKe94 Lajoie, R. and R.K. Keller. Design and reuse in object-oriented frameworks: patterns, contracts, and motifs in concert. Proceedings of the 62nd congress of the Association Canadienne Française pour l'Avancement des Sciences (ACFAS), Montreal, Canada, May 1994.

Lamp77 Lamport, L. Proving the correctness of multiprocess programs. IEEE transactions on software engineering, Vol. SE-3, No 7, pp. 125-143, March 1977.

Lamp94 Lamport, L. The temporal logic of actions. ACM transactions on programming languages and systems, Vol. 16, No 3, pp. 872-923, May 1994.

Lea96 Lea, D. Concurrent programming in Java. Addison-Wesley, London, 1996.

Lott90 Lott, C. M. Correctness is congruent with quality. ACM SIGSOFT, Vol. 15, No 5, pp. 19-20, October 1990.

MaCe99 Mandel, L and M. V. Cengarle. On the expressive power of OCL. Formal methods 1999, Vol. 1, LNCS 1708, pp. 854-874, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

Maes87 Maes, P. Concepts and experiments in computational reflection. Proceedings of OOPSLA '87, pp. 147-155, 1987.

MaPn81a Manna, Z. and A. Pnueli. Verification of concurrent programs: the temporal framework. In: The correctness problem in computer science (Eds. R.S. Boyer and J.S. Moore). International lecture series in computer science, Academic Press, London, pp. 215-274, 1981.

MaPn81b Manna, Z. and Pnueli, A. Verification of concurrent programs: temporal proof principles. In: Proceedings of the workshop on logics of programs (Yorktown-Heights, NY), Springer-Verlag Lecture Notes in Computer Science, 1981.

MC-Web Microsoft Corporation. Microsoft component services, server operating system, a technology overview. On-line Web white paper at http://www.microsoft.com/com/wpaper/compsvcs.asp.

MeSo99 Meyer, E. and J. Souquieres. A systematic approach to transform OMT diagrams to a B specification. Formal methods 1999, Vol. 1, LNCS 1708, pp. 875-895, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

Meye90 Meyer, B. Introduction to the theory of programming languages. Prentice-Hall, International (UK), 1990.

Meye97 Meyer, B. Object-oriented software construction, 2nd ed. Prentice-Hall, International, New Jersey, 1997.

Mikk98 Mikkonen, T. Formalizing design patterns. Proceedings of the 20th international conference on software engineering, IEEE Computer Society, pp. 115-124, 1998.

Misr99 Misra, J. A logic for the design of multiprogramming systems. South African Computer Journal (SACJ), No. 22, pp. 32 - 46, March 1999.

Mori90 Moriconi, M. Overview of the workshop. Proceedings of the ACM SIGSOFT International workshop on formal methods in software development, Napa, California, USA, 9-11 May, 1990. In: Software engineering notes, Vol. 15, Nr. 4, pp. viii-ix, September 1990.

Mosz86 Moszkowski, B.C. Executing temporal logic programs. Cambridge University Press, Cambridge, 1986.

OHE97 Orfali, R., D. Harkey and J. Edwards. Instant CORBA. John Wiley & Sons, Inc., U.S.A, 1997.

PaOs99 Paige, R. F. and J. S. Ostroff. Developing BON as an industrial-strength formal method. Formal methods 1999, Vol. 1, LNCS 1708, pp. 834-853, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

Parc90 ParcPlace Systems. Objectworks \ Smalltalk Release 4.1 user's guide. ParcPlace Systems, Inc., 1990.

Pnue77 Pnueli, A. The temporal logic of programs. Proceedings of the eighteenth symposium on foundations of computer science, Providence, RI, pp. 46-57, November 1977.

Pree94 Pree, W. Meta-patterns - a means for capturing the essentials of reusable object-oriented design. Proceedings of ECOOP '94, pp. 150 - 162, July 1994.

PvSK90 Parnas, D. L., A. J. van Schouwen and S. P. Kwan. Evaluation of safety-critical software. Communications of the ACM, Vol. 33, No. 6, pp. 636 - 648, June 1990.

RBPEL91 Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. Object-oriented modeling and design. Prentice-Hall, Inc., 1991.

RPS95 Ruiz-Delgado, A., D. Pitt and C. Smythe. A review of object-oriented approaches in formal methods. The Computer Journal, Vol. 38, No. 10, pp. 777 - 784, 1995.

RSC-Web Rational Software Corporation. On-line Web UML document set at http://www.rational.com/uml/references/notation_guide_ch$x$.html, where $1 \leq x \leq 10$.

Sand90 Sanders, B. Stepwise refinement of mixed specifications of concurrent programs. In: Proceedings of IFIP TC2/WG2.2/WG2.3 Working conference on programming concepts and methods, Sea of Galilee, Israel, April 1990. M. Broy and C. Jones (Eds). Elsevier Science Publishers B.V. Amsterdam 1990.

Seli93 Selic, B. An efficient object-oriented variation of the statecharts formalism for distributed real-time systems. Submitted to IFIP Conference on hardware description languages and their applications, April 26 - 28, 1993, Ottawa, Canada.

ShLa89 Shankar, A.U. and S.S. Lam. Construction of network protocols by stepwise refinement. REX workshop on stepwise refinement of distributed systems: models, formalisms, correctness, LNCS 430, pp. 669-695, Springer-Verlag Berlin, Heidelberg 1989.

Sifa99 Sifakis, J. Integration, the price of success. Formal methods 1999, Vol. 1, LNCS 1708, pp. 52-55, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

SiCh97 Sivilotti, P.A.G. and K.M. Chandy. A distributed infrastructure for software component technology. Technical report CS-TR-97-32, California Institute of Technology, September 1997.

Sivi97 Sivilotti, P.A.G. A method for the specification, composition and testing of distributed object systems. Ph.D. thesis, California Institute of Technology, CS-TR-97-31, December 1997.

Spiv92 Spivey, J.M. The Z notation: A reference manual, 2nd ed. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

Syke76 Sykes, J.B (ed). The concise Oxford dictionary of current English (6th ed.). Oxford University Press, Oxford, 1976.

Tane92 Tanenbaum, A.S. Modern operating systems. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

TMP99 Tyugu, E., M. Matskin and J. Penjam. Applications of structural synthesis of programs. Formal methods 1999, Vol. 1, LNCS 1708, pp. 551-569, Toulouse, France, September 20-24, 1999. J. Wing, J. Woodcock, J. Davies (Eds). Springer-Verlag Berlin, Heidelberg 1999.

Vilj95 Viljaama, P. The patterns business: impressions from PLoP-94. ACM Software engineering notes, Vol. 20, No. 1, pp. 74 - 78, January 1995.

Vino97 Vinoski, S. CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Communications magazine, Vol. 35, No. 2, pp. 46 - 55, February 1997.

Wing90 Wing, J. A specifier's introduction to formal methods. Computer, pp. 8-24, September 1990.

Wolp87 Wolper, P. On the relation of programs and computations to models of temporal logic. In: Temporal logic in specification (Ed. G. Goos and J. Hartmanis). Springer-Verlag, Berlin, 1987.

Yoko90 Yokote, Y. The design and implementation of concurrent Smalltalk. World Scientific Publishing Co. Pte.Ltd. Singapore, 1990.