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.

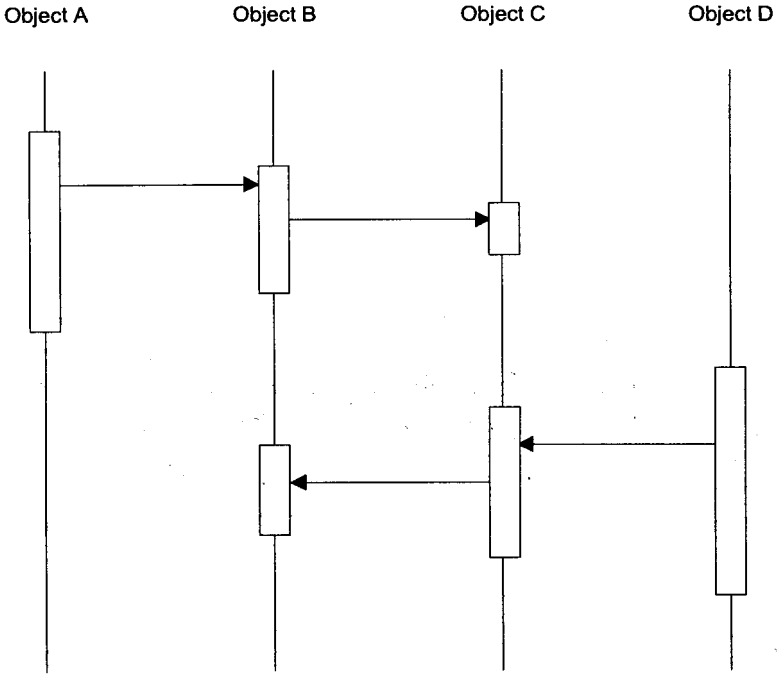

Figure 3-1(a). Synchronous operation invocations in a sequential program.

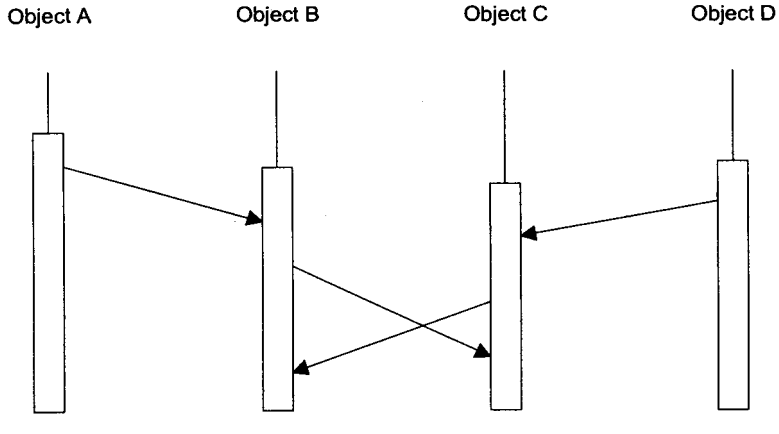

Figure 3-1(b). A scenario for deadlock during synchronous operation invocations.

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.

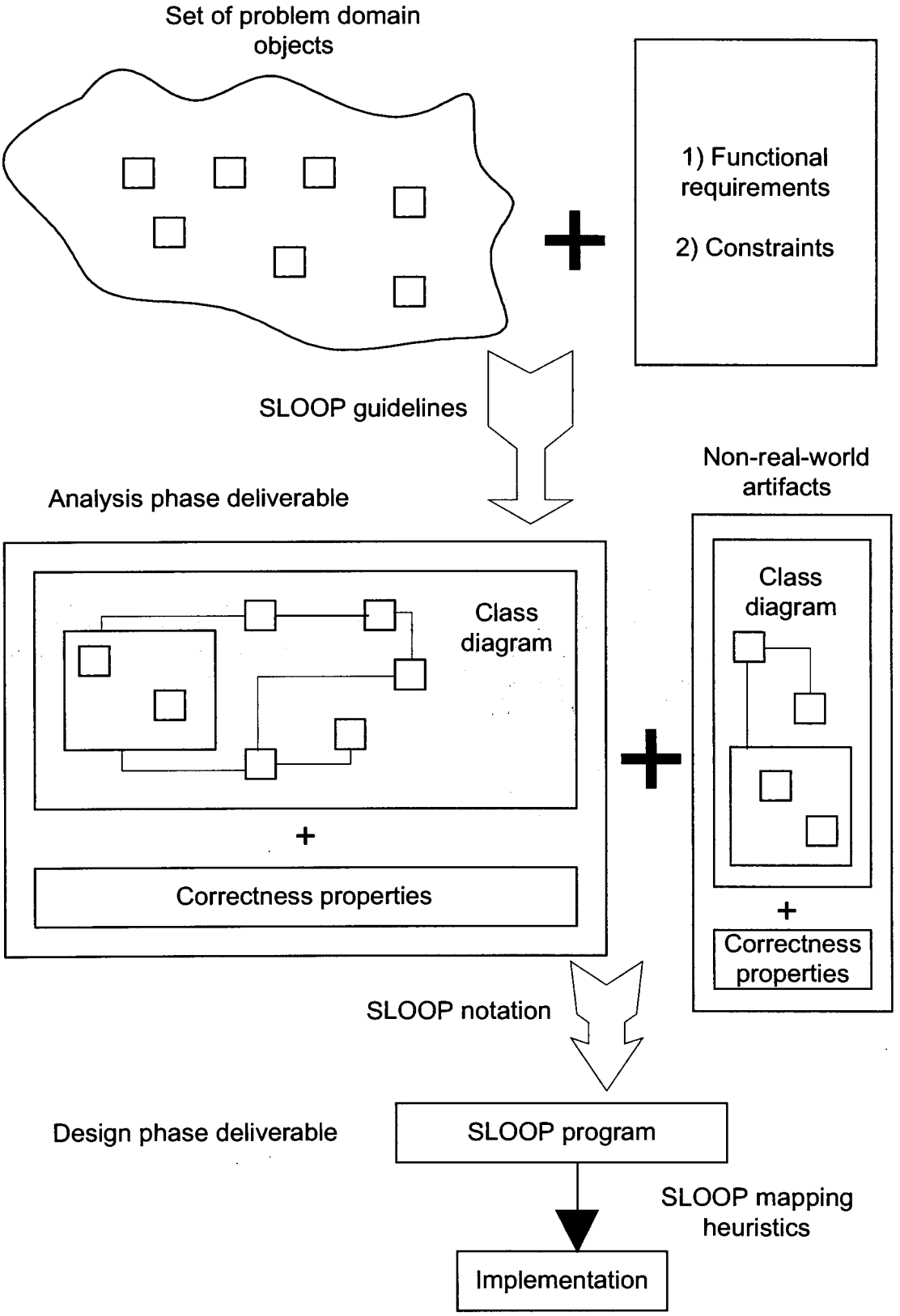

Figure 4-1. Overview of the SLOOP method.

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.

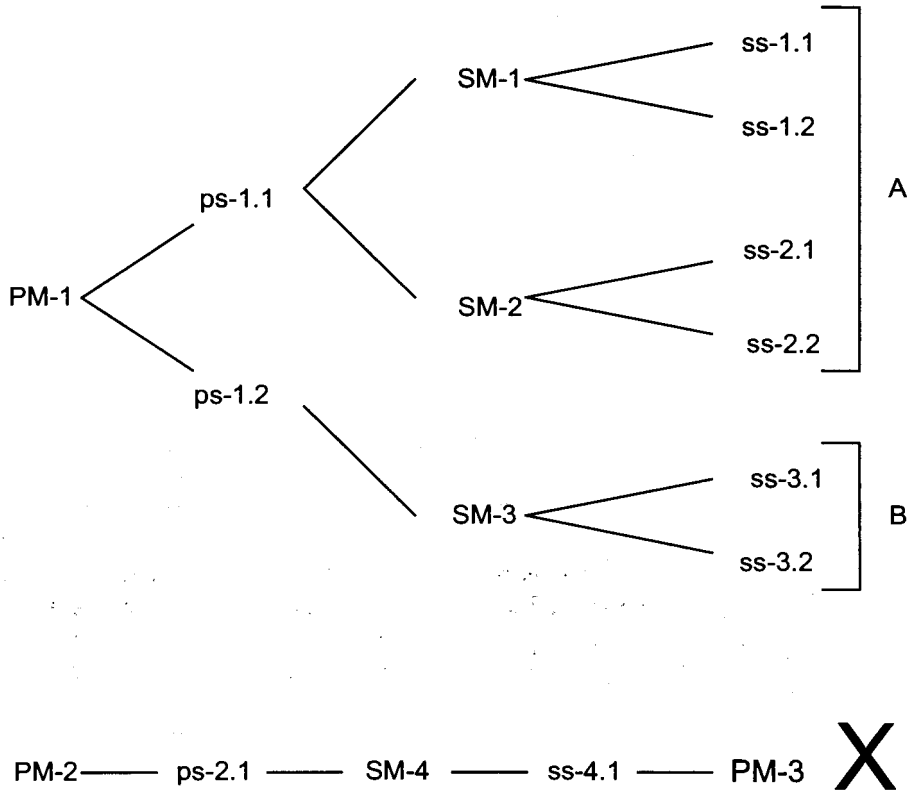

Figure 4-2. Scenarios representing atomic executions.

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.

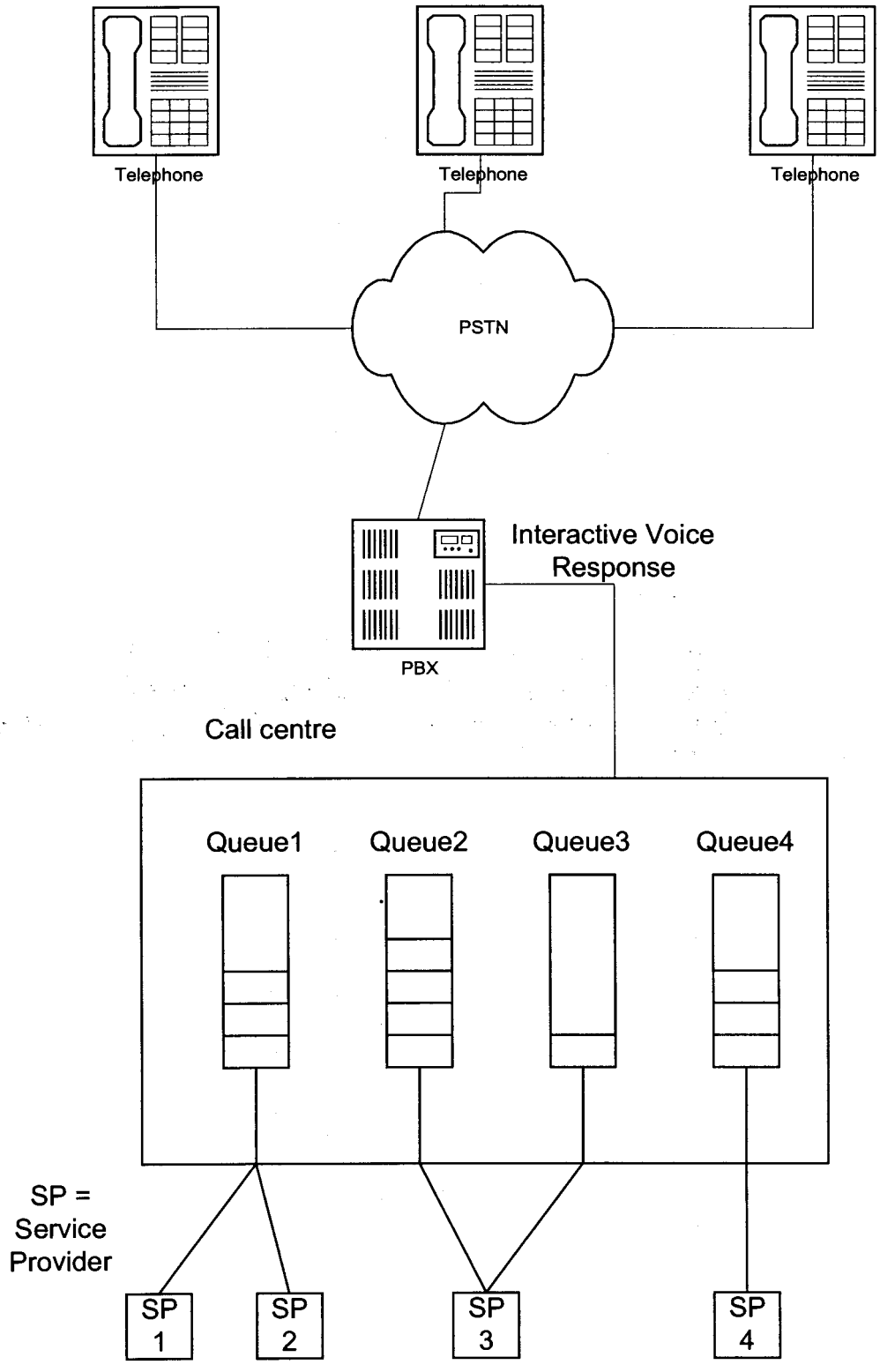

Figure 4-3. Call centre example.

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.

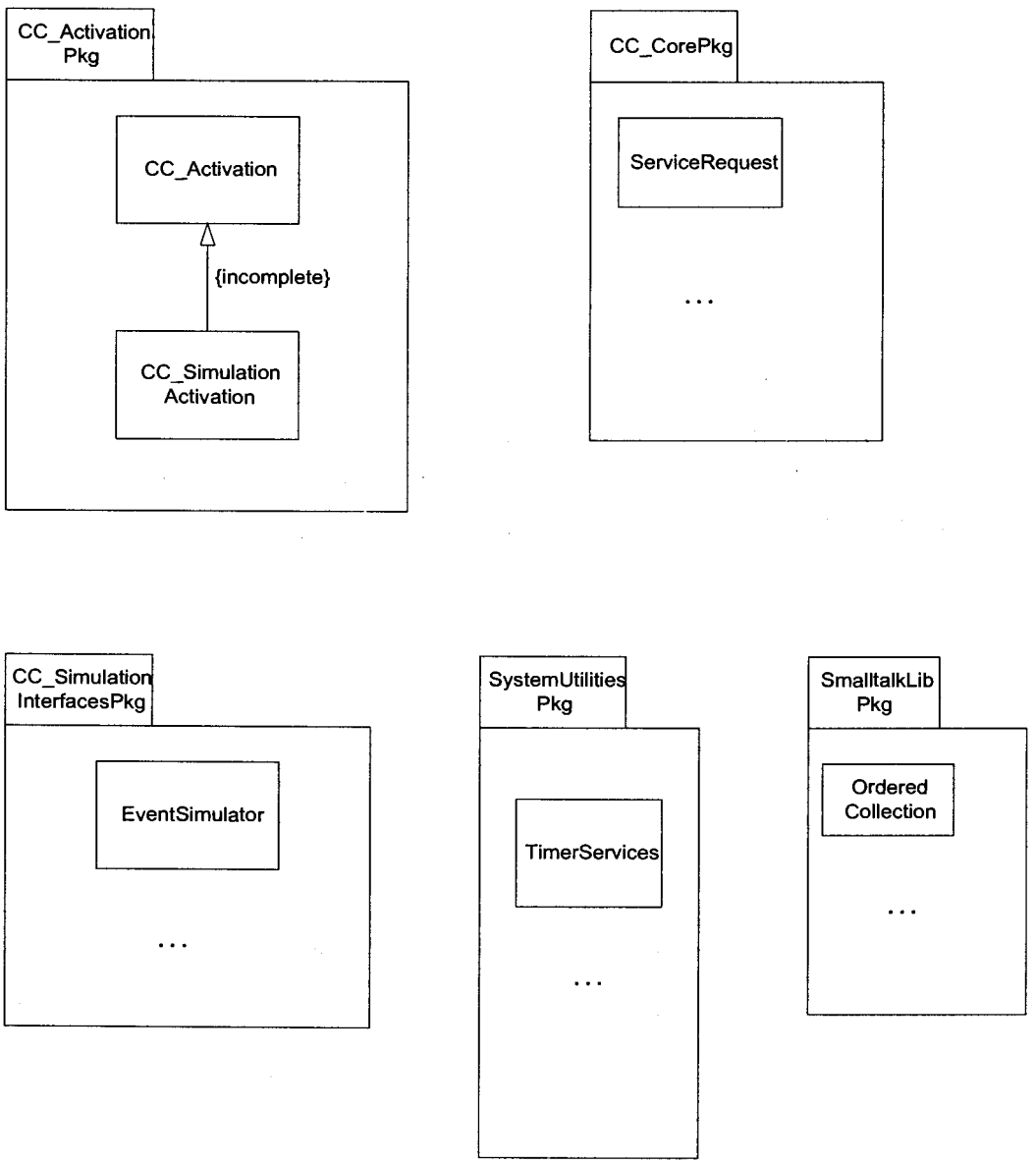

Figure 4-4. Package structure of the call centre.

[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**.

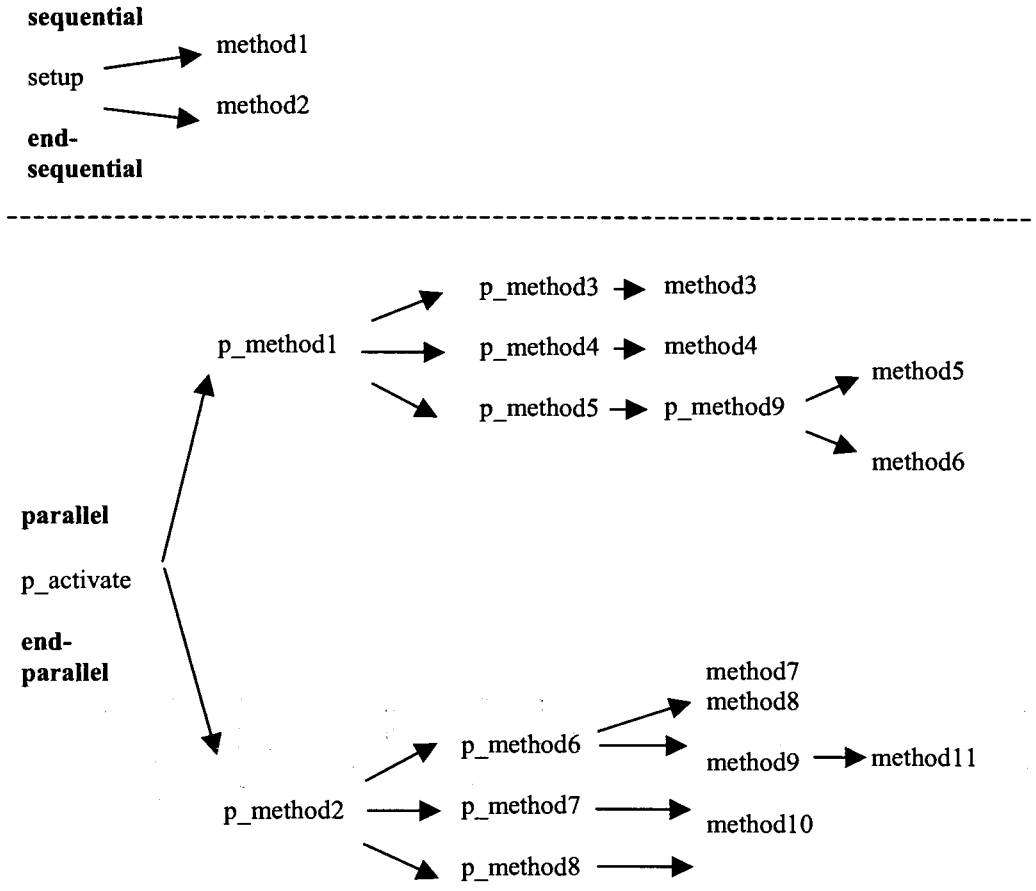

Figure 4-5. Example combinations of methods from different categories.

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.

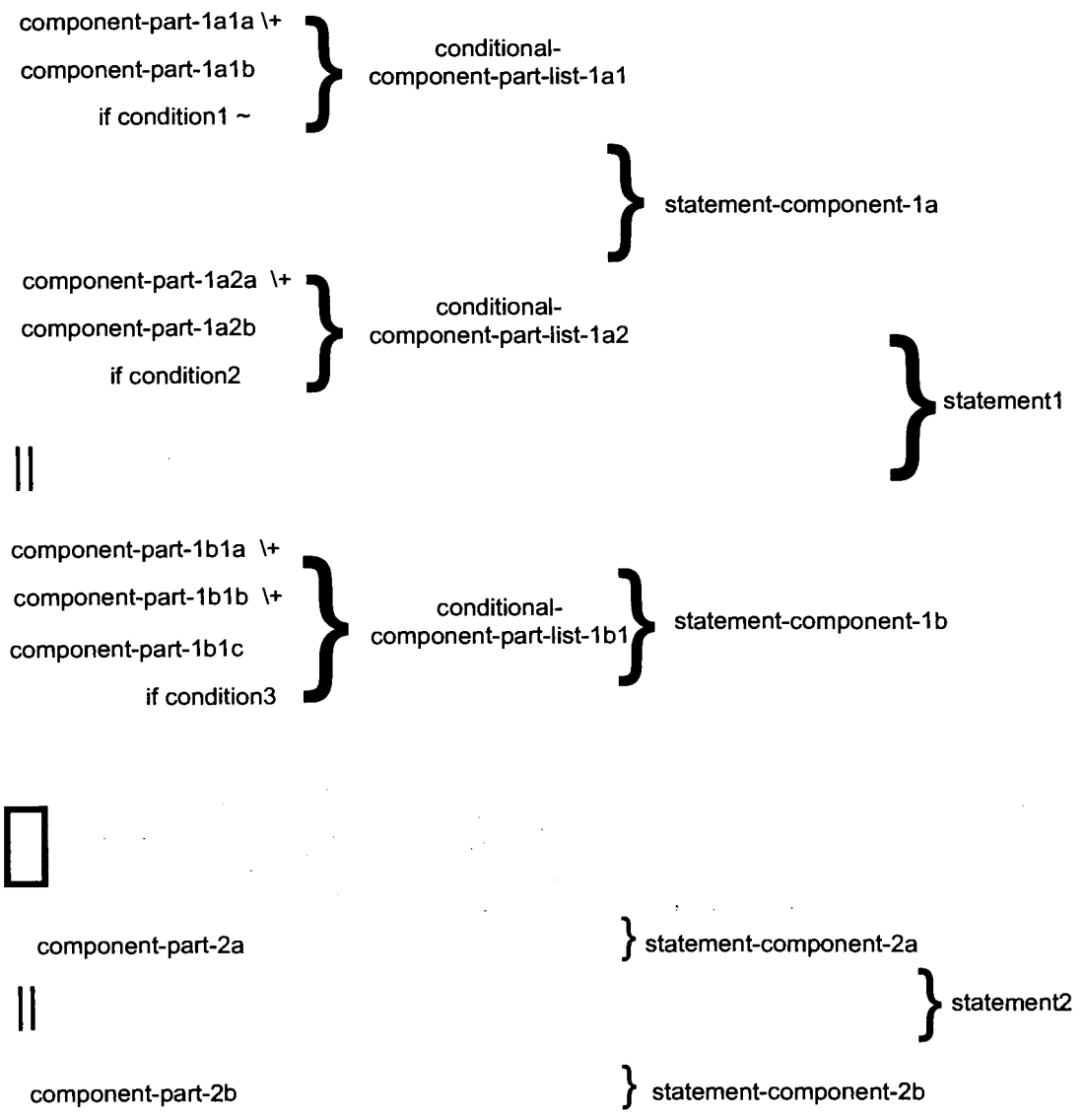

Figure 4-6. SLOOP statement structure.

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.

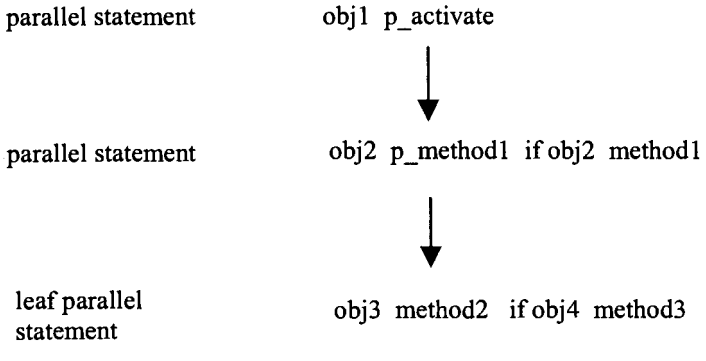

Figure 4-7. The role of *if* clauses in the parallel statement hierarchy.

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.

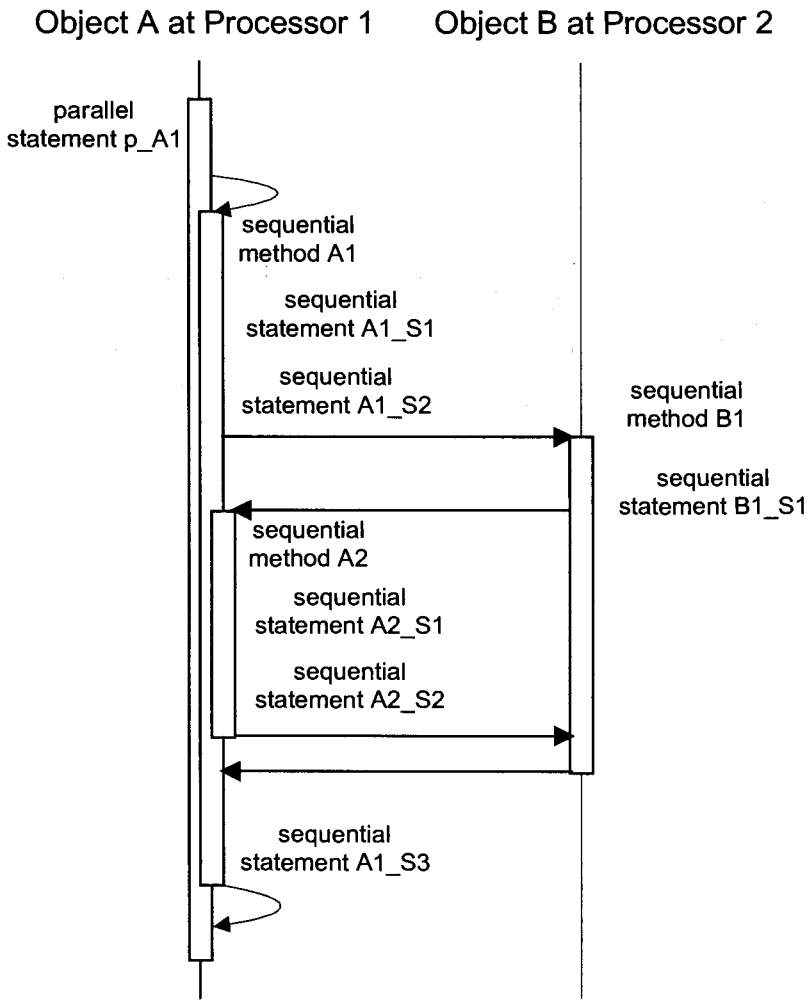

Figure 4-8(a). Cyclic invocation of the sequential methods of an object.

[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`.

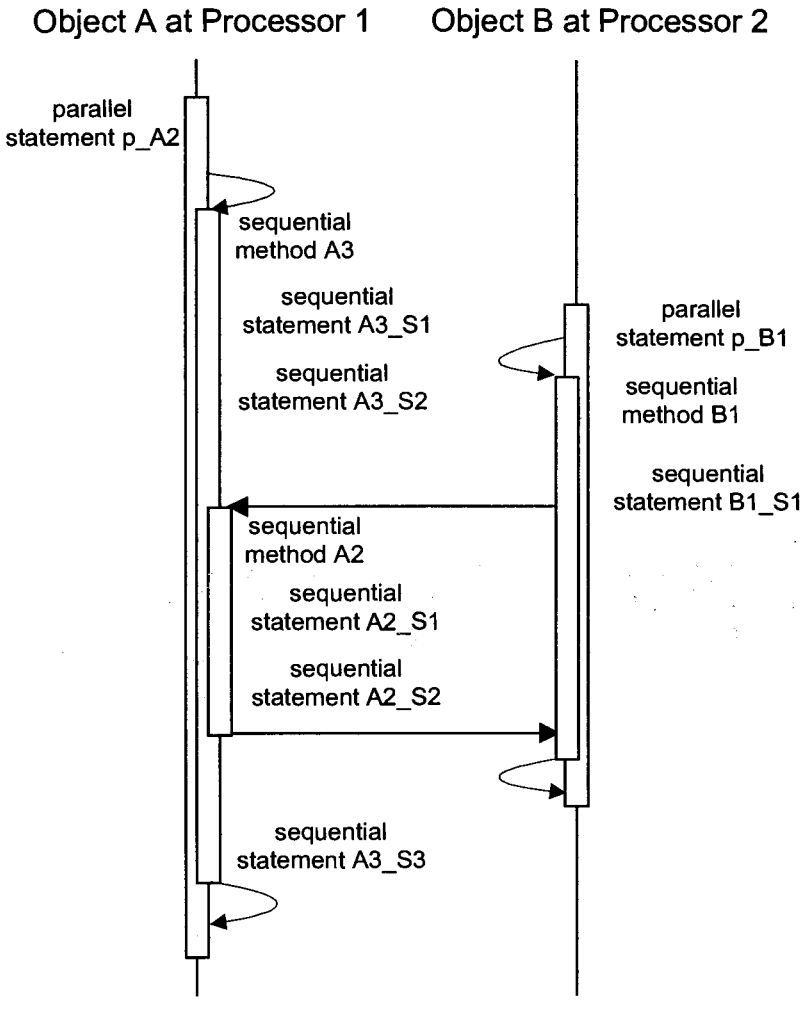

Figure 4-8(b). Example of an invalid execution sequence.

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.

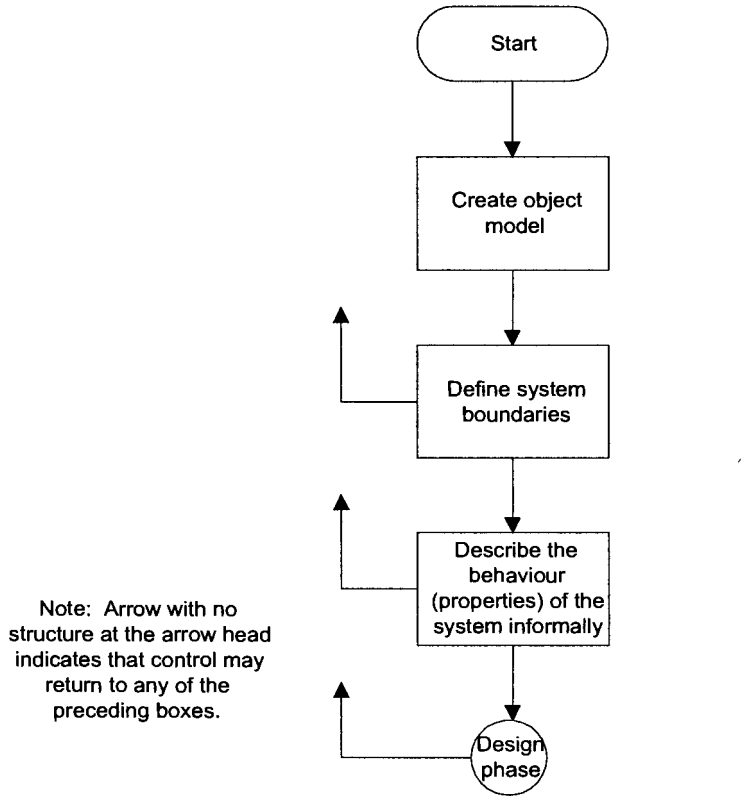

Figure 4-9. Requirements analysis phase steps.

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.

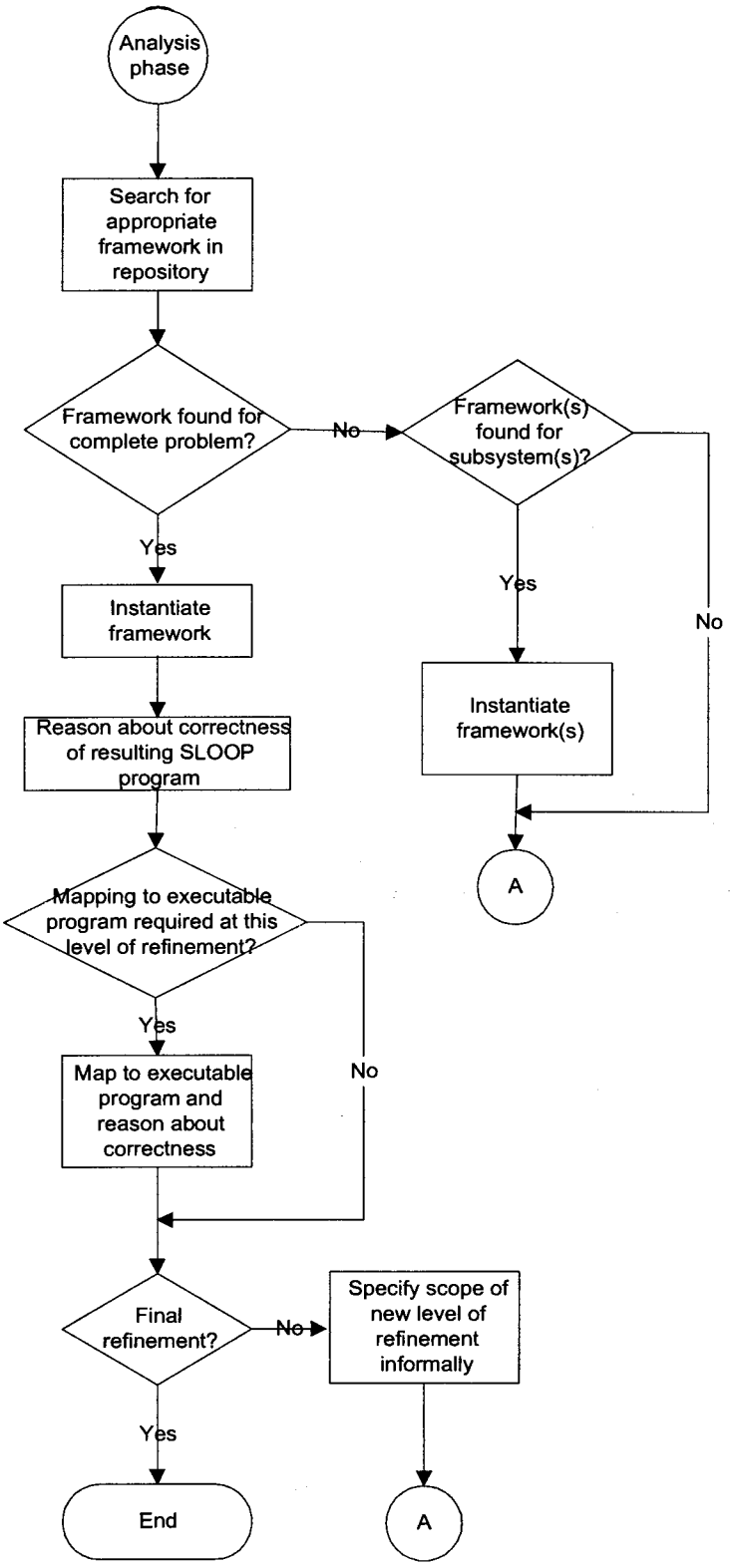

Figure 4-10(a). Design and implementation phases (Part 1 of 2).

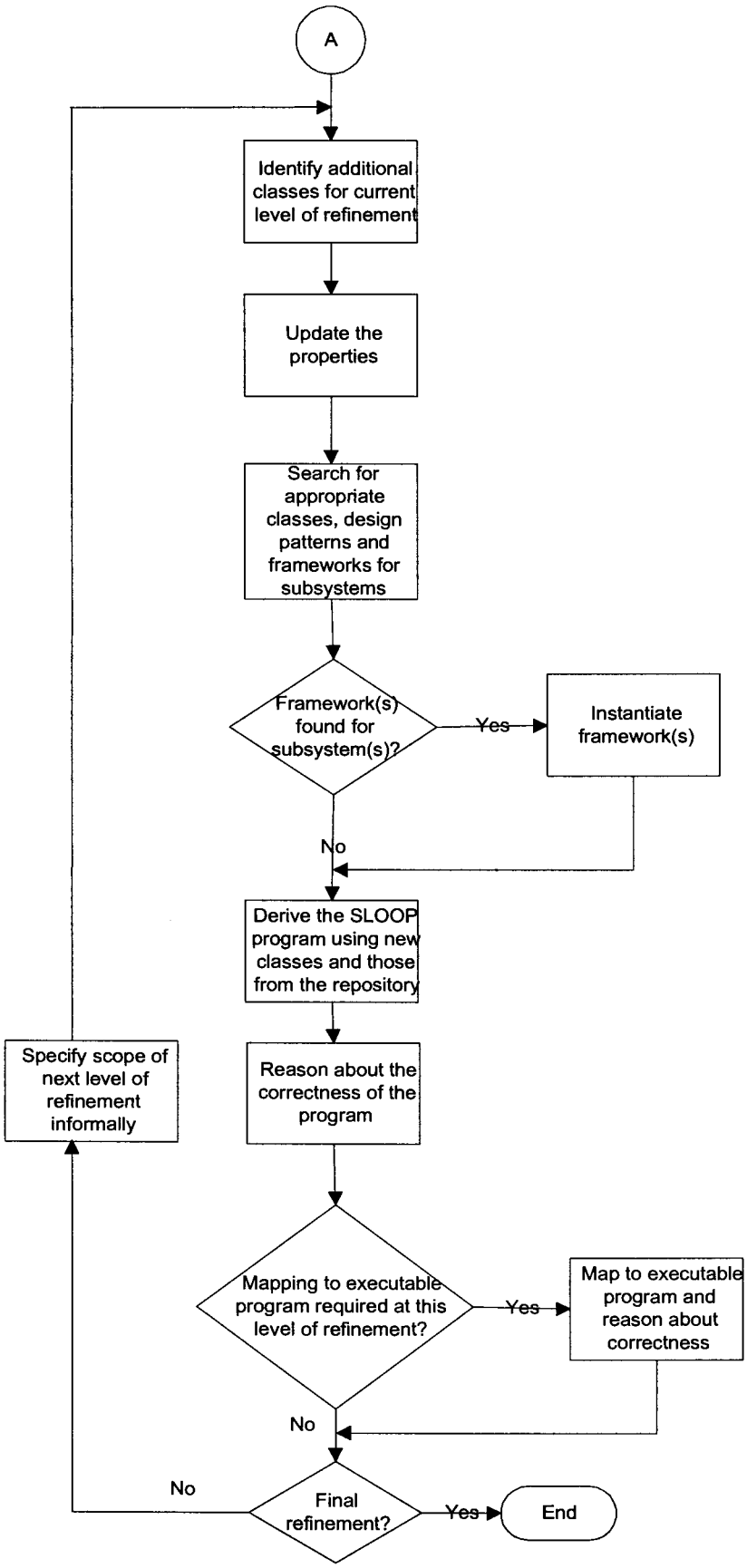

Figure 4-10(b). Design and implementation phases (Part 2 of 2).

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.