CHAPTER 7

REASONING ABOUT SLOOP PROGRAMS

7.1 Introduction

The assertions in a SLOOP program serve several purposes. Firstly they provide a means of conveying the **semantics** of the specified classes. Secondly they facilitate **reasoning about the correctness** of the program. Finally, they provide the necessary information to derive the SLOOP statements. All of these aspects are covered in this chapter. It is also shown how correctness reasoning benefits from two key characteristics of the SLOOP approach, viz. **reusability** and the **absence of control flow**.

7.1.1 Conveying the semantics

When a designer seeks information about the behaviour of a class, the assertions associated with a SLOOP class and its methods enable the designer to obtain this information without having to study the code. Section 7.2 describes the **nature** of the information that is obtained from the various parts of a SLOOP class specification.

7.1.2 Reasoning about correctness

Another important role of the assertions is to facilitate reasoning about correctness. Section 7.3 deals with this topic. An example of an informal proof of each **type** of correctness property as listed in Chapter 5, Section 5.2.4, is given.

Each example also serves to cover some **aspect** of correctness reasoning. For example, in one of the discussions a detailed account is given of the **steps** that are usually followed when an informal proof is presented. The use of **induction** [MaPn81b] when proving **invariance** properties and the use of **eventuality chains** [MaPn81b] when proving **liveness** properties are described. Other issues that are covered are, inter alia, the responsibilities of the client of an object and the significance of the `postconditions:` and `postconditions: withArguments:` constructs. Some of the examples presented in Section 7.3 deal with various reusability aspects of correctness reasoning and others cover the impact of the SLOOP computational model on correctness reasoning. These topics are discussed in more detail below.

7.1.3 Reusability

The fact that the system is designed as a set of classes suggests that the procedure used to reason about the correctness of the system should take advantage of this architecture. The aim is therefore to prove the properties (informally) on a **per class** basis and then rely on those

properties to hold when the methods of such a class are invoked. This approach **simplifies the arguments** and provides **structure** to the procedure.

In order to achieve this one needs to show that if each component behaves correctly in isolation, then it behaves correctly in concert with other components [AbLa89]. In UNITY, the **restricted union rule of superposition**[1] specifies that any statement *r* may be added to the underlying program provided that *r* does not assign to the underlying variables [ChMi88]. If program composition is performed according to this rule, every property of the underlying program is a property of the transformed program [ChMi88].

In the SLOOP method it is argued that when a new class is **added** to an existing set of classes in a SLOOP program, then the behaviour of the individual classes remains correct, provided the new class does not violate any of the preconditions specified for the methods of the existing classes. Since each class **encapsulates** its own data, the class itself controls the way in which this data may be modified. The **restricted union rule of superposition** is therefore used in SLOOP when new classes are added.

When **specialization** of a class is performed, the descendant class can modify the variables inherited from its ancestors, so this is similar to the concept of **union**[2] used in UNITY programs. The union of two UNITY programs *F* and *G* contains the statements from both programs and these statements can access and modify the same variables. The union theorem given in [ChMi88] specifies how **unless** and **ensures** properties, as well as **fixed points** are preserved under union[3].

The notion of a **conditional property** was included in [ChMi88] to deal with liveness properties of the form *p* **leads-to** *q* when considering the union of *F* and *G*. In short, a conditional property of program *F* consists of a **hypothesis** and a **conclusion**, each of which is an unconditional property. The hypothesis represents the specification of a system in which *F* can be embedded. The conclusion describes the effect of embedding *F* in the system. The union of *F* and *G* is the result of embedding *F* in the system.

When a SLOOP class is being specialized, new methods are added or existing methods are overridden. Thus, the original class can be viewed as *F* and the descendant can be viewed as the union of *F* and *G*, where *G* represents the statements added by the descendant. The constraints that must be satisfied by the descendant represent the hypothesis described above. These constraints are the class and method properties (excluding liveness properties of the form *p* **leads-to** *q*) of the descendant's ancestors. By using these properties as hypotheses, conclusions can be derived that are of the form *p* **leads-to** *q* and which represent the liveness properties of the ancestors. In turn, these liveness properties can be used to derive new liveness properties for the descendant.

Since the correctness properties of a descendant should not violate any correctness properties of its ancestors, it implies that the specialization of a class should not cause any of the existing

---

[1] The concept of superposition in the UNITY context was described in more detail in Chapter 2, Section 2.5.4.

[2] A brief discussion of the union concept was presented in Chapter 2, Section 2.5.4. More details are available in [ChMi88].

[3] The union theorem given in [ChMi88] is as follows:

1. p **unless** q in F [] G = (p **unless** q in F ∧ p **unless** q in G)
2. p **ensures** q in F [] G =
   [p **ensures** q in F ∧ p **unless** q in G] ∨
   [p **ensures** q in G ∧ p **unless** q in F]
3. (**FP** of F [] G) = (**FP** of F) ∧ (**FP** of G)

classes in the system to behave incorrectly. Thus, any class that sends messages to the descendant of the original class, can still rely on the pre- and postconditions of the original methods. The preconditions of the descendant's method will not be strengthened and postconditions will not be weakened. This topic is discussed further in Section 7.3.1.4.

The correctness properties specified for a class and its methods are the responsibility of that specific class. Each class is **obliged** to guarantee that the **postconditions** of each of its properties hold, **provided** the **preconditions** are met. A property is therefore only proved once for a given class and thereafter it may be **reused** by other classes as a lemma[4]. Whenever an object invokes a method of another object, the client object has to ensure that the preconditions of the method being invoked are met.

One of the features of a SLOOP program is the fact that the statements that are executed infinitely often are **encapsulated** in parallel methods and therefore **form part of the appropriate classes** comprising the system. Exactly the same principles that apply to the sequential methods of a class therefore also apply to its parallel methods, i.e. the correctness properties of the parallel methods can be reused in the same way as those of sequential methods.

The system itself can be viewed as a composite class. The correctness properties of this class represent the required behaviour of the system[5]. Once the correctness properties of the classes that form **part** of the system have been proved (informally), the behaviour of these classes may be assumed to be guaranteed as specified in these correctness properties. When reasoning about the correctness of the **system**, these assertions can be reused as lemmas.

The correctness properties of the composite class representing the system specify the **interactions** of the objects that form part of the system. When a composite class is subclassed in order to add a new class to the system, the effect of the resulting additional or modified object interactions must be taken into account in the correctness properties of the subclass of the composite class. This applies to any composite class; not only to the one representing the system under development

A class might **reuse** the correctness properties of its ancestors, **override** them or **add** new properties. The impact of the inheritance feature of the SLOOP method is discussed in Section 7.3.1.4.

Note that design patterns can also be reused and their correctness should also be shown. However, that topic will not be dealt with in this chapter. All the issues surrounding the usage of design patterns in the SLOOP method will be covered in Chapter 9.

7.1.4 Absence of control flow

The absence of control flow in the parallel methods allows one to reason in terms of the conditions, operations and assignments of **individual** parallel statements. There is no need to consider the various **combinations of statement interleavings** in a multi-process environment, since the order in which these statements are executed is irrelevant. The **sequential methods** are invoked from the parallel methods and are viewed as **terminating functions**. Each **parallel statement** is executed **atomically**. In the correctness arguments one can rely on the fact that each statement will be executed inifinitely often.

---

[4] A lemma is defined as an "assumed or demonstrated proposition used in argument or proof" [Syke76].

[5] In the call centre example, the CC_SimulationActivation class represents the behaviour of the system.

7.1.5 Using correctness properties to derive SLOOP statements

The methods of the various classes of a system are defined once the design phase correctness properties have been specified. The behaviour described by the correctness properties yields the necessary information to derive the methods of the classes. The liveness and/or precedence properties provide the basis for deriving the parallel methods, as will be seen in Section 7.4. The infinitely often execution of the statements of the parallel methods of the various classes has to result in the desired progress being made.

The sequential methods are also derived from the correctness properties. Some of them are even referenced directly in the correctness properties themselves, as will be shown in Section 7.4. Some correctness properties are used to refine statements that have been derived from other correctness properties. This role of correctness properties will also be demonstrated in Section 7.4.

The above-mentioned features of the SLOOP method are now exemplified using the call centre example introduced in the earlier chapters.

Note: The examples below are at a detailed level. In many cases the discussions of the issues at hand are interspersed with various parts of the informal proofs. In order to highlight the points being made, the relevant parts of the text appear in boxes.

7.2 Conveying the semantics

As stated earlier, one of the purposes of specifying the correctness properties of a class and its methods is to enable the designers who wish to reuse the class to ascertain the behaviour of the class without having to study the code. This is illustrated by the discussion of the `ServiceProviderSimulator` class below. The purpose of this class is to simulate the behaviour of a service provider. The class was first mentioned in Section 5.3.2 and its analysis level properties were identified in Section 5.4.5.2. In Section 6.2.3 the `EventSimulator` class was introduced as a parent class for simulator classes. Section 6.5 contained a summary of the methods of the `EventSimulator` and `ServiceProviderSimulator` classes after the design level refinements had been performed. The full SLOOP specifications of the `EventSimulator` and `ServiceProviderSimulator` classes appear in Appendix B, Sections B.5 and B.13 respectively.

7.2.1 The static nature of a class

A designer seeking information on **the static nature** of the `ServiceProviderSimulator` class should refer to the **list of class and instance variable names of the class and its ancestors**. This list represents the **attributes** of the class and if the class is a **composite** class, some of these variables will refer to the **parts** of the composite class. The `ServiceProviderSimulator` class inherits the following variables from the `Event=Simulator` class:

```
rand
        "This variable refers to an instance of the Random class from
        the Smalltalk library.  The instance is created when the
        EventSimulator subclass is instantiated.  The instance of the
        Random class maintains a seed from which the next random number
        is generated. The random number is used to start a timer with a
        random value."
```

```
newEventRequired
        "When the value is equal to true it means that a new event is
        required.  Once the variable has been set to true, a random
        timer will be started at some point afterwards.  When the timer
        is started, newEventRequired is set to false.  It is the
        responsibility of the subclass to set this variable to true when
        a new event is required, since each subclass will have its own
        conditions for requiring a new event.  Once the timer expires,
        an event will be generated, as will be described in the comments
        section of the generatingEvent variable."
currentRandomTimeoutValue
        "This variable contains the value of the random timeout
        currently being requested.  The purpose of this variable is to
        provide a mechanism for referencing the current timeout value in
        the correctness arguments.  Note that the SLOOP statements could
        therefore have been rewritten without this variable while still
        providing the same functionality.  However, in that case it
        would not have been possible to formalise certain correctness
        properties (such as DL1-04(EventSimulator), listed in Appendix
        B, Section B.5)."
generatingEvent
        "The value is equal to true if the timer has expired and an
        event has to be generated, otherwise it is equal to false.  The
        subclass sets this variable to false at the time when the event
        is generated.  The actual event that is generated is also the
        reponsibility of the subclass, since each subclass will generate
        a different type of event."
timerOutstanding
        "This variable is set to true when a timer is started and it is
        set to false when a timeoutElement is removed from the
        timerEventQ (i.e. when an expired timer has been processed).
        The purpose of this variable is to provide a mechanism for
        reasoning about the uniqueness of outstanding timers in the
        EventSimulator class.  In this class only one timer requested by
        the EventSimulator may be outstanding at a time.  The
        timerOutstanding variable is used in the preconditions of the
        startRandomTimer:withMaximum: method as well as in the
        postconditions of the resetTimerExpired: method.  If subclasses
        need to support multiple simultaneous timers, then the
        preconditions of the startRandomTimer:withMaximum: method need
        to be weakened and the postconditions of the resetTimerExpired:
        method need to be strengthened. Since the purpose of the
        timerOutstanding variable is to facilitate correctness
        reasoning, the SLOOP statements could have been rewritten
        without this variable while still providing the same
        functionality."
timerId
        "This variable contains the identifier of the timer currently
        being requested.
```

From the above description the designer learns that the `EventSimulator` is a composite class, since the `Random` class forms part of it. The purpose of each variable is also gleaned from the comment following each variable name. As pointed out in the relevant comments, some variables are introduced solely for the purpose of reasoning about the correctness of the class.

For example, the statements

```
  currentRandomTimeoutValue :=
      (self nextRandomNumber: maximumValue)
[] aTimerServices start: self id: timerId for:
      currentRandomTimeoutValue
```

could have been replaced by

```
[] aTimerServices start: self id: timerId for:
        (self nextRandomNumber: maximumValue)
```

thereby eliminating the `currentRandomTimeoutValue` variable from the SLOOP statements in the `startRandomTimer:withMaximum:` method. However, a similar replacement would then have been required in the correctness properties referencing this variable. Unfortunately this is not possible, since the `(self nextRandomNumber: maximumValue)` expression has side-effects and may therefore not be used in correctness properties. The rationale for this restriction is the fact that correctness properties are at a **meta-level,** i.e. they describe the behaviour of the system and should **never modify** the system state.

The usage of the `timerOutstanding` variable is shown in Appendix B, Section B.5.

The `ServiceProviderSimulator` subclass adds a number of variables to the above list:

```
instance variable names
serviceRequest
        "This variable refers to the service request currently being
        serviced by the service provider simulator.  Note that the
        reference to the ServiceRequest instance is passed to the
        simulator as a parameter, i.e. the ServiceRequest instance is
        not created by the ServiceProviderSimulator instance and
        therefore does not form part of it."
serviceProviderCategory
        "This variable contains the name of the service provider
        category to which the service provider simulator belongs."
categoriesServed
        "This is an ordered collection containing the names of the
        service request categories serviced by this service provider.
        The purpose of this array is to facilitate a round robin
        servicing scheme of these categories. That prevents starvation
        of a specific service category."
nrOfCategoriesServed
        "This variable contains the number of service request categories
        serviced by this service provider. It is used in the calculation
        when the categoryIndex is updated."
categoryIndex
        "This variable is used as index into the categoriesServed
        collection. It is used to determine the next service request
        category to be serviced by this service provider. It is
        incremented modulo nrOfCategoriesServed. Its values range from 0
        to nrOfCategoriesServed - 1"
```

The purpose of the `categoriesServed`, `nrOfCategoriesServed` and `categoryIndex` instance variables is to prevent a `ServiceProviderSimulator` instance from ignoring one of the service queues that it should be servicing for ever. The `ServiceProviderSimulator` class is a composite class (it inherits the `Random` class from its `EventSimulator` parent class). The `categoriesServed` instance variable contains a reference to another part of the composite class, viz. `OrderedCollection`.

7.2.2 The dynamic nature of a class

To find out about the **dynamic behaviour** of a service provider simulator, one would have to become acquainted with the various **properties** of a **class and its ancestors**. The properties listed below are the **class** properties of the parent of the `ServiceProviderSimulator` class, namely the `EventSimulator` class. Subsequently the **method** properties of the `EventSimulator` class will be examined, followed by the inspection of the `ServiceProviderSimulator` **class and method** properties. Note that the **analysis level**

properties are only specified **informally**. Where **possible,** the **design level properties** are specified **formally**. The rationale for only giving informal specifications of some of the design level properties is given later.

```
class properties
        "Liveness"
        "When a simulation event is required, a simulation timer is eventually started."
                                                    "AL2-01 (EventSimulator)"

        "Liveness"
        "If a simulator timer expires, the simulator eventually has to generate an event."
                                                    "AL2-02 (EventSimulator)"

        "Clean behaviour"
        <∀ anObject ::
                    invariant anObject class ~~ EventSimulator
        >                                           "DS2-01 (EventSimulator)"
        "The EventSimulator class is an abstract class and should not be instantiated."

        "Clean behaviour"
        invariant    rand notNil ∧ rand class = Random
                                                    "DS2-02 (EventSimulator)"
        "Once rand has been initialized to refer to an instance of the Random class, it is never
        set to nil while the instance of the EventSimulator subclass exists."

        "Clean behaviour"
        "The currentRandomTimeout value is always within the range specified by the
        precondition of the start:id:for: method of the TimerServices class."
                                                    "DS2-03 (EventSimulator)"

        "Global invariant"
        "All outstanding timers requested by an EventSimulator subclass instance are identified
        uniquely with respect to the requestor."
                                                    "DS3-01 (EventSimulator)"
```

The properties are organised first according to the development phase from which they originated and next according to the property type. The analysis level properties are presented first, since they provide the designer with the gist of the functionality of the class. Properties *AL2-01* and *AL2-02* convey to the designer that a timer will be started if a new event is required and when that timer expires, an event will be generated.

The last four properties are design level safety properties. By inspecting them, the designer is able to learn about the design of the `EventSimulator` class. The first property reveals that the `EventSimulator` class is an abstract class. The second property guarantees clean behaviour as far as the composite and its parts are concerned: it ensures that subclass instances of the `EventSimulator` class will never send messages to a non-existing instance of the `Random` class.

The next clean behaviour property specifies that the method to start a new timer will always be invoked with a requested timer value that is within the range as specified in the precondition of the method being invoked. The **reason why this property is not specified formally at this point** is because it refers to the requirements of another class. The **reference to this other class** is passed as an **argument** to the `p_simulate:timeoutEventsIn:` method of the `EventSimulator` class and the formal specification of this property is therefore given within that method. The interested reader is referred to Appendix B, Section B.5 for details.

The fourth property is a global invariant which ensures that the notification of the expiry of an `EventSimulator` timer can be correlated with the correct timer request. It specifies that each outstanding timer requested by a subclass instance of the `EventSimulator` class is identified uniquely with respect to its requestor. This property is specified **informally only** in this example.

There are numerous ways in which this property can be specified formally. One possibility is formulate it in terms of the currently outstanding timers and the identifiers associated with them. That implies that several additional instance variables would have to be defined and maintained. It is **debatable whether the additional complexity is justified** in the case of the `EventSimulator` class, where only one timer may be outstanding at a time. Should a subclass require multiple simultaneous outstanding timers, the modifications required to support those timers might also, as a side-effect, yield the necessary additional variables to facilitate a formal version of the above property. For these reasons, it was decided that the informal version of property *DS3-01* would suffice for the `EventSimulator` class.

As stated earlier, the identification of correctness properties during a specific software development phase is **not** an **ad hoc** process. The checklist as presented in Section 5.2.4 is used to aid the software designer in following a **structured** and **systematic** approach when first recording these properties. When they are subsequently inspected in order to evaluate the class for potential reuse in other systems, different designers may prefer different groupings of the properties. One possibility is to group the properties according to functionality. For example, for the `EventSimulator` class all properties related to starting a timer could be grouped together and similarly all properties related to the expiry of a timer could be grouped together. Combining the properties in a different way is especially useful during the familiarisation process.

However, the SLOOP method favours the **recording** of the properties based on the checklist as discussed above. One of the reasons for this preference is that it addresses one of the problems associated with the conjunctive nature of logic-based methods, viz. the **completeness** of the specification [JiZh96]. Although one cannot guarantee the completeness of such a specification [Fran92], at least one is guided to consider each property type and to reflect whether or not there are any useful properties that can be specified for each type.

In this example the next step is to inspect the properties of the **individual methods** of the **superclass** (i.e. the `EventSimulator` class). That is followed by the inspection of the **class** properties of the **subclass** (i.e. the `ServiceProviderSimulator` class), after which the properties of the `ServiceProviderSimulator` **methods** are examined. However, the order in which these steps are performed is not prescribed by the SLOOP method.

The following methods are defined for the `EventSimulator` class:

```
initialize
nextRandomNumber:
startRandomTimer:withMaximum:
timerExpired:
resetTimerExpired:
p_simulate:timeoutEventsIn:
```

A great deal can be learnt from the correctness properties of these methods. For brevity, this information is summarised here. The interested reader is referred to Appendix B, Section B.5 for the SLOOP specification of the correctness properties of the above methods.

The `initialize` method is executed when an `EventSimulator` subclass is created. It ensures that all the instance variables of the `EventSimulator` class will have initial values. The descendants may perform some additional initialisation, but the total correctness property of

the `initialize` method guarantees that at least the `EventSimulator` variables will not be uninitialised.

The `nextRandomNumber:` method always returns a number ranging from 1 to `maximumValue`, where `maximumValue` is passed to the method as an argument. The `startRandomTimer: withMaximum:` method guarantees that a timer is started with a value within the range from 1 to the maximum value received as one of the arguments of the method. This method invokes the `nextRandomNumber:` method in order to generate the timer value.

The `timerExpired:` method returns true if a timer requested by the `EventSimulator` subclass instance has expired and false if not. This is determined by the presence or absence of the relevant `timeoutElement` in the `timerEventQ`. The `resetTimerExpired:` method removes a `timeoutElement` from the `timerEventQ`.

Finally, the liveness properties specified for the `EventSimulator` class are realised via its parallel method. The `p_simulate:timeoutEventsIn:` method ensures that a timer is started if `newEventRequired` is true and it guarantees that `generatingEvent` will be set to true once the timer has expired.

It is clear from these properties that the `EventSimulator` class has nothing to do with the actual event that is generated. It also does not determine when a new event is required. These are functions of the subclasses. In order to find out **when a new is event is required** and **what type of event is generated** in the case of a `ServiceProviderSimulator` class, the properties of that subclass are examined next.

class properties

```
<∀ categoryIndex where categoryIndex ≥ 0 ∧
categoryIndex ≤ nrCategoriesServed - 1 ::
invariant serviceRequest notNil ⇒ ¬self canAcceptNextSR:
      (categoriesServed at: (categoryIndex + 1))
>                          "AS3-01 (ServiceProviderSimulator)"
```

"A service provider simulator services a single service request at a time."

```
"If a service request is currently assigned to the simulator, no
other service request from any of the categories being served by
this simulator will be served by the latter."

serviceRequest isNil ∧ ¬newEventRequired unless
serviceRequest notNil ∧ newEventRequired
                          "AS4-01 (ServiceProviderSimulator)"
```

"When a new service request is assigned to the service provider simulator then a new service provider simulator event is required."

"Note: The parent class, viz. EventSimulator, contains a parallel method which monitors the value of newEventRequired. If it detects that newEventRequired is true, it starts a timer and sets newEventRequired to false."

```
serviceRequest notNil ∧ ¬newEventRequired unless
serviceRequest isNil ∧ ¬newEventRequired
                          "AS4-02 (ServiceProviderSimulator)"
```

"If a service request is assigned to the service provider simulator and newEventRequired is false, then newEventRequired remains false for at least as long as the service request is still assigned to the service provider simulator."

```
"This has the effect that this simulator will not start another
timer before the servicing of the current service request has
been completed."

generatingEvent ∧ serviceRequest notNil ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil ∧
      ¬generatingEvent   "AP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the service provider simulator terminates the connection currently associated with it and becomes available to service a new service request."

"Note: The parent class, viz. EventSimulator, contains a parallel method which sets generatingEvent to true when a timer has expired."

```
<∀ aServiceRequest where serviceRequest = aServiceRequest ::
 serviceRequest = aServiceRequest ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil
>                          "AP1-02 (ServiceProviderSimulator)"
```

"A service request remains assigned to a service provider simulator until the latter completes the service and terminates the connection."

```
"Clean behaviour"
invariant categoryIndex ≥ 0 ∧
      categoryIndex < nrOfCategoriesServed
                           "DS2-01 (ServiceProviderSimulator)"
```

"The categoryIndex is always greater than or equal to zero and less than nrOfCategoriesServed."

```
"Clean behaviour"
invariant    categoriesServed notNil ∧
             categoriesServed class = OrderedCollection
                           "DS2-02 (ServiceProviderSimulator)"
```

"Once categoriesServed has been initialized to refer to an instance of the OrderedCollection class, it is never set to nil while the ServiceProviderSimulator instance exists."

```
<∀ categoryIndex where 0 ≤ categoryIndex ∧
categoryIndex < nrOfCategoriesServed ::
¬(self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1)))
leads-to
      self canAcceptNextSR:
      (categoriesServed at: (categoryIndex + 1))
>                          "DL2-01 (ServiceProviderSimulator)"
```

"For any service category serviced by the service provider simulator, the service provider simulator will eventually be able to service a request from that service category."

Property *AS4-01* (the first **unless** property) reveals how `newEventRequired` is set to true, viz. when a new service request is assigned to the simulator. The statements of the parent class ensure that a timer is started if `newEventRequired` is true. When the timer is started, `newEventRequired` is set to false.

The `newEventRequired` variable then remains false until the service has been completed. The latter happens once the timer has expired and the service request is deallocated from the simulator. This is evident from properties *AS4-02*, *AP1-01* and *AP1-02*. Properties *AS3-01* and *AP1-02* guarantee that only one service request is serviced at a time. Property *AP1-01* provides information regarding the nature of the event that is generated. Property *DS2-01* specifies the allowed values of `categoryIndex`, property *DS2-02* guarantees that the `categoriesServed` variable will not be nil and property *DL2-01* ensures that the service provider simulator will service each service category supported by it.

The properties of the individual methods are now inspected in order to obtain more information about the ServiceProviderSimulator class. These methods are as follows:

```
startSimulation:using:
moreInit:using:
registerServiceProvider:using:
serviceProviderCategory
serviceRequest
canAcceptNextSR:
processServiceRequest:
p_generateEvent
p_updateCategoryIndex:
```

Details of the correctness properties of the methods listed above are provided in Appendix B, Section B.13. For the purposes of this discussion it suffices to point out that the first three methods are used when a `ServiceProviderSimulator` instance is created. These methods ensure that the `serviceRequest` variable is set to nil, that a service provider category is assigned to the simulator and that the simulator is registered with the relevant service categories during instance creation. These methods are also responsible for creating the `categoriesServed` ordered collection and for recording all the service categories supported by this `ServiceProviderSimulator` instance in that collection.

The `serviceProviderCategory` and `serviceRequest` methods are accessing methods that return the category of the service provider simulator and the current service request being serviced respectively.

The `processServiceRequest:` method is invoked by the client of the `ServiceProviderSimulator` instance in order to assign a new service request to the simulator. The client has to ensure that the precondition of the `processServiceRequest:` method holds when it invokes it. The `canAcceptNextSR:` method forms part of the precondition. It returns true if the `serviceRequest` variable is equal to nil and the message argument matches the next service category to be serviced by this simulator. It returns false otherwise. By using the `canAcceptNextSR:` method in the precondition of the `processServiceRequest:` method, it can be guaranteed that a new service request is only accepted if no other service request is currently being serviced by this simulator. It also ensures that the service categories are serviced in a round robin fashion.

The postconditions of the total correctness property of the `processServiceRequest:` method specify that `newEventRequired` will be true and `serviceRequest` will not be nil when the method has completed its execution. Thus it ensures that `newEventRequired` is set to true when a new service request is assigned to the simulator. In turn, the parallel method of the `EventSimulator` class ensures that a timer is started if `newEventRequired` is true. Once the timer expires, `generatingEvent` is set to true (also via the parallel method of the parent class). The precedence property of the `p_generateEvent` parallel method then guarantees that the relevant event is generated if `generatingEvent` is true.

The `categoryIndex` instance variable is updated in the `processServiceRequest:` and `p_updateCategoryIndex:` methods. As is evident from the correctness properties of these methods, the value is updated in a way which ensures that the relevant service queues can be serviced in a round robin fashion by this simulator. The `processServiceRequest:` method ensures that the `categoryIndex` is updated whenever a new service request is accepted from a service queue associated with the service category indicated by the current value of `categoryIndex`. The parallel method, `p_updateCategoryIndex:`, checks the service queue associated with the service category indicated by the current value of `categoryIndex`. If that service queue is empty, the `categoryIndex` is also updated.

In summary, information about the **static** nature of a class is obtained via the class and instance **variables** of the specified class and its ancestors. The software designer needs to inspect the **correctness properties** of the class and its methods, both of the class itself and of its ancestors, in order to learn about the **dynamic behaviour** of the class.

The purpose of the discussion in this section was merely to demonstrate how **information about the behaviour** of the class could be obtained from the correctness properties specified for the class and its methods. No correctness arguments were given. In the sections that follow, it will be demonstrated how one can use informal correctness arguments to show how the correctness properties of the relevant individual methods ensure that various correctness properties of the class are satisfied. That description forms part of a discussion of the impact of various features of the SLOOP method on correctness reasoning. It is the topic of the next section.

7.3 The impact of various SLOOP features on correctness reasoning

The purpose of this section is to illustrate how various features of the SLOOP method result in a **gain in simplicity** as far as correctness reasoning is concerned. Examples of informal arguments to reason about safety, liveness and precedence properties are given next. For each property **type** listed in Chapter 5, the correctness arguments for at least one property are presented. The software designer would use such arguments during the software development process to verify informally that already stated correctness properties indeed hold.

Note that the SLOOP method **does not mandate** that these correctness arguments form part of the official documentation of a project. One could therefore choose to adopt this style of thinking without recording these informal proofs. However, by documenting the correctness arguments they can be checked by others and they are then also available to those who might want to reuse the classes to which the correctness arguments apply.

Apart from illustrating how one can reason about different types of correctness properties, the specific properties in the examples below are also chosen to highlight additional issues. For example, in Section 7.3.1.2 it is shown how correctness reasoning is simplified due to the fact that **program location counters** can be **ignored** when the **atomic** units of execution (the parallel statements) are considered. In Sections 7.3.2.3 and 7.3.3.2 it is illustrated how the distinctive properties of the **leads-to** and **ensures** relations respectively are utilised in correctness arguments.

In Chapter 5 the desired analysis level properties of the call centre system were stated. These properties describe the interactions of the objects that comprise the system and are captured within the `CC_SimulationActivation` class and its superclass, `CC_Activation`. The system properties listed in Chapter 5 were written in terms of the analysis phase artifacts, i.e. prior to the design level refinements. For example, during the analysis phase the

`ConnectionContainer` class was defined as the container class of the `Connection` instances. As a result of the design phase refinements, it was found that the `ConnectionContainer` class was superfluous. Instead, the `Array` class from the Smalltalk library sufficed. The correctness properties that include the design level refinements refer to this `Array` instance as `userConnections`.

The correctness properties as discussed in the remainder of this chapter include the design level refinements. Note that properties that emanated from the analysis phase retain their identifiers as assigned to them during that phase. Thus, even though design phase refinements have been added, their identifiers do not change, since their origin (i.e. the analysis phase) remains the same.

The correctness properties of a class may be overridden in a descendant, as will be demonstrated in Section 7.3.1.4. In that case the property in the subclass has the same identifier as the one in its ancestor, but the identifier is followed by the name of the **ancestor** in brackets. When referring to such a property **from within another class**, the words "in *class-name*" have to be added, where *class-name* is the name of the class which overrides the property.

7.3.1 Safety properties

In this subsection the safety properties relating to clean behaviour, global invariants and the unless relation are considered. Each subsection sheds light on something specific:

- Section 7.3.1.1: the advantages of using **macros** with respect to correctness reasoning;
- Section 7.3.1.2: the influence of the **SLOOP computational model** on correctness reasoning and the impact of the object-oriented features such as **data encapsulation** and **reusability** on correctness arguments;
- Section 7.3.1.3: the **allocation of responsibilities** regarding the preconditions during method invocations; and
- Section 7.3.1.4: the **effect of inheritance** on correctness arguments.

The generic approach towards proving a correctness property informally can be described as follows: First of all the correctness property is specified formally. Then the strategy to be followed in order to prove informally that the property holds is determined. Thereafter the correctness arguments are given. The latter can be presented from first principles (i.e. by inspecting the statements of the SLOOP program), or correctness properties that have already been proven, can be reused if applicable. The above procedures are described in detail in Section 7.3.1.2.

7.3.1.1 Using the correctness arguments of a clean behaviour property to illustrate how the use of macros in SLOOP programs can simplify correctness reasoning

The purpose of this section is to illustrate the **advantages of using macros** in SLOOP programs with respect to correctness reasoning. This is achieved by discussing the correctness arguments of property *AS2-01(CC_Activation)*, **an analysis level clean behaviour safety property**. It is one of the safety properties of the `CC_Activation` class and specifies the following:

invariant `userConnections capacity = maxConn ∧ maxConn > 0`
"AS2-01 (CC_Activation)"

*"The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer."*

This property is used to describe the capacity restrictions of the call centre. The purpose of this property was discussed in detail in Chapter 5, Section 5.4.1.2. In order to show that the above property always holds for the `CC_Activation` class and its subclasses, the contents of this property first needs to be examined more closely.

First of all, it sends the `capacity` message to `userConnections`, the `Array` instance. One therefore needs to look at the characteristics of the `Array` class and its methods. The `capacity` method returns the number of indexed instance variables of the `Array` instance. The indexed variables are used to hold the elements of the `Array` instance. When an `Array` instance is created, all its indexed instance variables are created. The number of indexed instance variables remains fixed throughout the existence of the `Array` instance. It is equal to the value of the parameter that is passed to the `Array` class when the `new:` method is invoked.

In order to show **informally** that the capacity of `userConnections` is equal to `maxConn` and does not change, it needs to be shown that `userConnections` is created as an `Array` instance of this capacity. This is done by inspecting the SLOOP program statements as given in Appendix B. The `userConnections Array` instance is created in the `initialize` method of the `CC_Activation` class[6] via the statement below:

```
userConnections := SmalltalkLibPkg:::Array new: maxConn
```

It is the only statement that assigns a value to the `userConnections` variable in the `CallCentreSimulation` program. Thus, this statement ensures that the `capacity` of the `userConnections` object is equal to `maxConn` and it therefore means that `userConnections` can hold exactly `maxConn` number of elements.

The second part of property *AS2-01* states that `maxConn > 0` is an invariant. This part of the example highlights the **advantages** of using **macros** in a SLOOP program. There are a number of references to `maxConn` in the `CC_Activation` class, but `maxConn` is not an attribute of this class, as is evident from the list of instance variables of this class in Appendix B, Section B2. Instead, all of these references are expanded to `config maximumConnections`, the message expression which obtains the value of *maximumConnections* from an instance of the `Configuration` class. One of the invariants of the `Configuration` class[7] is the following:

```
<∀ ( t, u, v, w) where
t > 0 ∧ u > 0 ∧ v > 0 ∧ w > 0 ::
invariant
        maximumConnections = t ∧
        maximumServiceCategories = u ∧
        maximumServiceProviders = v ∧
        maximumAllowableTimeout = w
>                                          "DS2-01(Configuration)"
```

Thus, the value of `maximumConnections` is always greater than zero and it also does not change. Since property *DS2-01* is an invariant of the `Configuration` class, it means that once the class is instantiated, this property holds for that class. All other classes that send the `maximumConnections` message to the `Configuration` instance may therefore assume that it will return a value that is greater than 0, since it is guaranteed by the `Configuration` class.

---

[6] The `initialize` method of the `CC_Activation` class is presented in Section B.2 of Appendix B.

[7] The Configuration class is specified in Appendix B, Section B.4. The rationale for having such a class was given in Chapter 6, Section 6.3.1.

The **advantage of using a macro** rather than an instance variable in the above case is evident from the fact that **correctness property *DS2-01* of the `Configuration` class may be reused**. Since a *macro-variable* may not appear on the left-hand side of any SLOOP statement, it can never contain any value other than the one to which its associated *macro-expression* evaluates.

One alternative to the above approach is for each client class to define an instance variable representing the maximum number of connections. It is then the responsibility of each class to ensure that this variable always contains the correct value (which implies that a correctness property stating this has to be defined for each client class and the correctness of this property also needs to be shown).

This example gives an **indication** of the **nature** of the correctness arguments that are used in the SLOOP method. It also gives an **indication** of the **level of rigour** that is present in these arguments. The next section provides more detail regarding the steps that are followed during correctness reasoning in the SLOOP method.

7.3.1.2 Demonstrating how the computational model and object-oriented features such as data encapsulation and reusability influence the approach followed during correctness reasoning

Another clean behaviour property of the call centre system, viz. property *AS2-04* defined in Chapter 5, Section 5.4.1.2, is now used to illustrate the general approach followed during informal correctness reasoning in the SLOOP method.

The purpose of this section is twofold, viz.

- Firstly, it demonstrates how the **computational model** influences the correctness arguments. It is shown how **induction** is used when proving **invariance** properties informally.
- Secondly, it illustrates how object-oriented features such as **data encapsulation** and **reusability** are taken advantage of in the correctness arguments.

Informally, property *AS2-04(CC_Activation)* is defined as follows:
If a connection is terminated, it implies that its associated service request is not present in the input queue.

The approach has three major steps:

- The first step is to specify the property formally. (It is difficult to reason about something that can be interpreted in different ways.) Thus, the property has to have an unambiguous meaning.
- The strategy required to arrive at the property as conclusion, is determined.
- Arguments are presented to support the various claims as outlined in the strategy.

Specifying the correctness property formally

In order to specify property *AS2-04* more formally, one needs to consider how a terminated connection should be represented. In Section 6.3.2 it was explained that a design decision was made to create the maximum number of `Connection` instances upon startup and to use the state of an instance to determine whether it represented an unassigned connection or not.

The rationale for making this decision was given in that section. Briefly, it ensures that the parallel statements of all `Connection` instances are always present to be selected for execution. As stated in [ChMi88], it would be difficult to define a fair execution rule for statements if the set of program statements changed dynamically. If `Connection` instances could be created and destroyed dynamically after initialization, then it would imply that their associated methods would only be present during the existence of the instances.

When the analysis level property *AS2-04 (CC_Activation)* was formulated in Chapter 5, the aim was to avoid any design level detail. As a result, there were no references to the `state` of a `Connection` instance or the presence or absence of `Connection` instances in the `userConnections` collection in order to indicate the availability of a connection. The terminology used in the specification of the property was at a very high level of abstraction.

Once the design decision had been made to reflect the availability of the `Connection` instance via its `state`, the property was rewritten as follows:
If a connection is idle, it implies that its associated service request is not present in the input queue.
More formally:

```
<∀ aConnection where userConnections include: aConnection ::
      invariant    aConnection isIdle  ⇒
                   ¬(inputQ includes: (aConnection serviceRequest))
>                                                  "AS2-04 (CC_Activation)"
```

Below correctness arguments are given to show that property *AS2-04(CC_Activation)* is indeed invariant.

The strategy to be followed for the correctness arguments

The correctness arguments in methods based on the conventional computational model take the location counters of the processes involved in the computation into account. For example, in [MaPn81a] the proof principles presented for establishing correctness properties clearly take cognisance of the location counters. In the SLOOP correctness arguments that are employed at the design level, there is no notion of concurrent processes and therefore they **do not contain references to location counters** of different processes. Instead, the correctness arguments show that there **exist** statements in the program that will ensure that the specified **progress** properties will hold and also that **no statements exist** that will violate the **safety** properties.

The following statements of the `CallCentreSimulation` program are therefore considered:

- The **sequential statements** in the *activation-section* of the program (in the order of their appearance).
- All the **parallel statements** that are activated directly or indirectly via the *activation-section* of the program (in any order).

The **sequential methods invoked by the above statements** are also taken into account.

The principle of **induction** [MaPn81b] is used when proving **invariants** in the SLOOP method. Considering the SLOOP computational model, this means that one has to show that the property holds initially, as well as after the execution of any parallel statement of the program.

The correctness arguments for property *AS2-04(CC_Activation)* are presented in two parts. Part A shows that the property holds initially. Part B contains the arguments to prove informally that the property also holds after the execution of each parallel statement of the program.

The arguments in part B are in support of two claims, viz.:

B1) The connection state changes to **not** 'IDLE' whenever the service request is added to the input queue.
B2) The state does not change to 'IDLE' while the service request is still present in the input queue.

Thus, while a service request is present in the input queue, the associated connection is not 'IDLE'. This implies that if a connection is 'IDLE', it cannot be present in the input queue, which is what invariant *AS2-04(CC_Activation)* specifies.

The correctness arguments

The correctness arguments for property *AS2-04* are now presented. In part A it is shown that the property holds immediately after initialization. In part B1 arguments are presented to prove informally that any statement that adds a service request to the input queue also changes the state of the connection to not 'IDLE' at that point. In part B2 it is shown that no statements exist that will change the state of the connection to 'IDLE' while the associated service request is still present in the input queue.

Part A:

In order to prove informally that property AS2-04(CC_Activation) holds **immediately after initialization**, the **sequential** statements in the *activation-section* are scrutinised. In the `CallCentreSimulation` program there is only one statement in the sequential part as can be seen in this SLOOP program excerpt:

```
program CallCentreSimulation
      sequential
      aCCSimulationActivation :=
            CC_ActivationPkg:::CC_SimulationActivation setup
      end-sequential
      parallel
      aCCSimulationActivation p_activate
      end-parallel
      "Packages"
      ...
end-program
```

The sequential statement sends the `setup` message to the `CC_SimulationActivation` class, which results in a new instance being created and initialized. The sequential methods that are invoked as part of the initialization are listed in Sections B.2 and B.3 of Appendix B. Inspection of these methods reveals that the `inputQ` is created via the following statement in the `initialize` method of the `CC_Activation` class:

```
inputQ := SmalltalkLibPkg:::OrderedCollection new: maxConn
```

However, there are no statements in the sequential methods that are invoked as part of the initialization that add service requests to the `inputQ`. Immediately after initialization the `inputQ` therefore contains no service requests.

Each `Connection` instance is created via the following statement in the `initConnection:` method of the `CC_Activation` class:

```
^CC_CorePkg:::Connection setup: index
```

As can be seen from the statements of the `initialize:` method[8] which is invoked as a result of the execution of the above statement, each `Connection` instance is in the 'IDLE' state after initialization.

Thus, initially each `Connection` instance is in the 'IDLE' state and the `inputQ` contains no service requests, which means that property *AS2-04(CC_Activation)* is satisfied.

Part B:

The next step is to prove informally that property *AS2-04(CC_Activation)* is preserved by the **parallel** statements of the program. The **amount of effort required is reduced** by the fact that only those parallel statements that are **relevant** to this property need to be considered. This is because each parallel statement forms part of a method that belongs to a class, and as a result of **data encapsulation**, the effect of the execution of a parallel statement is limited to the state of the objects that are known within that class instance. As a result only a limited number of statements need to be examined at a time. The steps described below therefore take advantage of the **structuring capabilities** inherent in an object-oriented method.

Part B1:

In order to show that the connection state changes to **not** 'IDLE' whenever the service request is added to the input queue, all the statements that add a service request to `inputQ` need to be found. The *activation-section* contains only one parallel statement, viz.

```
aCCSimulationActivation p_activate
```

In turn, the `p_activate` method[9] of the `CC_Activation` class contains several parallel statements:

```
"p_activate method of the CC_Activation class"
parallel
        self p_executeCPAgent
        "The parallel methods of the commsAgent are not invoked
        directly, but rather via the p_executeCPAgent method of the
        CC_Activation class."

        [] timer p_runTimer: timerEventQ
        "Activate the parallel methods of the timer object.  The timer
        parallel statements have the following functionality: Whenever a
        timeout occurs, the TimeoutElement instance representing the
        timeout is added to the end of the timerEventQ, which indicates
        to the requestor that the specified timer has expired."

        [] self p_categoriseAndAllocate
        "The parallel methods of the scAllocator object are invoked via
        the p_categoriseAndAllocate method of the CC_Activation class.
        The scAllocator parallel statements have the following
        functionality: Once a service request has been categorised, it
        is removed from the inputQ and appended to the appropriate
        serviceQ."
```

[8] The Connection class is specified in Appendix B, Section B.7.
[9] The p_activate method is specified in Appendix B, Section B.2.

```
        [] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
                    p_execute
          >
        "Activate the parallel methods of the ServiceCategory instances.
        Their parallel statements have the following functionality: For
        each service category the associated service queue and set of
        service provider agents are monitored.  If the service queue is
        not empty and a service provider agent in the spSubset
        associated with the service category is available to process a
        new service request, the first element of the service queue is
        removed and assigned to a service provider agent."

        [] < [] i where 1≤i≤maxConn :: self p_executeConnection:
              (userConnections at: i)
          >
        "The p_executeConnection method of the CC_Activation class is
        executed for each Connection instance in order to invoke the
        parallel methods of the latter.  The parallel statements of the
        Connection instances have the following functionality: When a
        connection has entered the 'TERMINATING' state, the
        communication provider agent is requested to terminate the
        connection.  Once all the procedures have been completed to
        terminate the connection, the connection and its associated
        service request are reset to their initial states."

        [] < [] k where 1≤k≤maxSP :: self p_executeSPAgent:
              (spAgentContainer at: k)
          >
        "The parallel methods of the service provider agents are not
        invoked directly, but rather by executing the p_executeSPAgent
        method of the CC_Activation class for each of the service
        provider agents."
end-parallel
```

These statements can be inspected in any order. For brevity, only the statement containing the `p_executeCPAgent` message is discussed here, since it is the one leading to the statement that assigns service requests to the `inputQ`. The `p_executeCPAgent` method is the responsibility of the subclass, which is the `CC_SimulationActivation` class[10] in this case. In the latter, this method contains the following statements:

```
"p_executeCPAgent method of the CC_SimulationActivation class"
parallel
        commsAgent p_simulate: timer timeoutEventsIn: timerEventQ
      [] commsAgent p_generateEvent: userConnections target: inputQ
end-parallel
```

The second parallel statement in this method invokes the `p_generateEvent:target:` method of the `CommsProviderSimulator` class[11], which contains the only statement in the `CallCentreSimulation` program that adds a service request to the `inputQ`:

```
"p_generateEvent:target: method of the CommsProviderSimulator class"
parallel
         inputQ addLast: (idleConnection serviceRequest) \+
         idleConnection assign
            if generatingEvent and: [idleConnection notNil] ~
         Transcript show: 'All connections busy'
            if generatingEvent and: [idleConnection isNil]
```

[10] The CC_SimulationActivation class is presented in Appendix B, Section B3.
[11] The CommsProviderSimulator class is specified in Appendix B, Section B.6.

```
        || newEventRequired := true \+
           generatingEvent := false
              if generatingEvent
end-parallel
```

The `p_generateEvent:target:` method of the `CommsProviderSimulator` class makes use of the `idleConnection` macro, which is defined in the `p_generateEvent:target:` method as:

```
idleConnection ≡ self getIdleConnection: userConnections
```

In turn, the `getIdleConnection` sequential method of the `CommsProviderSimulator` class is defined as:

```
^userConnections detect: [:each | each isIdle] ifNone: [nil]
```

Finally, the `isIdle` method of the `Connection` class[12] is defined as:

```
^state = 'IDLE'
```

Thus, the `p_generateEvent:target:` method has the following functionality: If an event has to be generated, an idle connection is searched for by checking whether there is any `Connection` instance in the `userConnections` array that is in the 'IDLE' state. If one is found, the associated service request is added to the `inputQ` and the state of the connection is changed to indicate that it is connected to a service user. The `assign` method of the `Connection` class contains the following statement:

```
state := 'CONNECTED'
      if state = 'IDLE'
```

If an idle connection cannot be found, a status message is generated.

From the statement in the `p_generateEvent:target:` method it is clear that a service request may only be added to the `inputQ` if the associated connection is in the 'IDLE' state. In a **single** multiple-assignment statement the service request is added to the `inputQ` and the state of the connection is changed to 'CONNECTED'. **Note that it is crucial that the insertion of the service request into the `inputQ` and the modification of the state of the connection to 'CONNECTED' should occur simultaneously, i.e. as part of the same atomic statement. This is essential for the correctness arguments of the above property, since it has to be shown that the service request cannot be added to the input queue while the connection remains in the 'IDLE' state.** This concludes Part B1 of the correctness arguments.

Note that in the above discussion it was shown how to locate a specific statement (i.e. one that adds an element to the `inputQ`) manually. In practice, any editing or browsing tool can be used to search for a statement which invokes a specific method, which makes it very simple to isolate the statements that need to be considered.

Part B2:

For Part B2 it has to be shown that the connection does not enter the 'IDLE' state while the service request is in the input queue.

The main arguments comprising Part B are as follows:
B2.1) The only transition to the 'IDLE' state is from the 'TERMINATING' state.
B2.2) At this level of refinement the method that results in a transition to the 'TERMINATING' state is not invoked while the service request is in the `inputQ` (i.e. the connection is not aborted or rejected). Only the service provider simulators invoke this method.

[12] The Connection class is specified in Appendix B, Section B.7.

B2.3) A service provider simulator only invokes this method if the service provider simulator has a service request assigned to it.
B2.4) Once a service request has been assigned to a service provider simulator the service request is no longer present in the `inputQ` and after a service request has been removed from the input queue, it remains outside the queue until the connection has been terminated and has reached the 'IDLE' state.

Part B2.1:

In this part it needs to be shown that the only transition to the 'IDLE' state of a connection is from the 'TERMINATING' state. The informal correctness arguments for B2.1 take advantage of some of the object-oriented characteristics of the SLOOP method, viz. its **structuring** and **data encapsulation** features. Instead of inspecting each statement of the SLOOP program, one first checks whether it is not possible to restrict the number of statements that need to be examined.

This is done by inspecting the `Connection` class in order to find out whether it provides any methods enabling clients to set its `state` instance variable to 'IDLE'. The idea is that if no method is provided to clients to set the state to 'IDLE', **the number of statements that need to be inspected is reduced** from all the statements in the program to only those of the `Connection` class.

The first thing that is discovered is the fact that there is no `state:` method[13] which allows a client to set the `state` to any value. Instead, the `Connection` class controls the values of the `state` instance variable by only providing an `assign` and a `terminate:` method to modify the value of `state`. The `assign` method sets the value to 'CONNECTED' and the `terminate:` method sets it to 'TERMINATING'.

Although the `initialize:` and `p_doWrapUp` methods set the value of `state` to 'IDLE', these are private and parallel methods respectively. A private method is not accessible to clients, while a parallel method cannot be invoked from within any sequential method (this rule was specified in Chapter 4, Section 4.2.3). The `p_doWrapUp` method is only invoked from within the `CC_Activation` class in order to **activate** the parallel statements of that method. Thus, the `Connection` instance itself is the only object that can set `state` to 'IDLE'. The only statement that sets `state` to 'IDLE' after instance creation and initialization, is the following one in the `p_doWrapUp` method:

```
"p_doWrapUp method of the Connection class"
parallel
      state := 'IDLE' \+
      serviceRequest reset \+
      currentHandlerInformed := false
            if currentHandlerInformed
end-parallel
```

The precondition for this method is that the instance variable `currentHandlerInformed` should be true. This variable is an instance variable of the `Connection` class that cannot be modified by any client. The only statement that sets it to true is in the `p_informCommsProvider:` parallel method of the `Connection` class and one of the preconditions for that method is that the state should be equal to 'TERMINATING', as can be seen below:

[13] The SLOOP specification of the methods of the Connection class can be found in Appendix B, Section B.7.

```
"p_informCommsProvider method of the Connection class"
parallel
      commsAgent terminate: self cause: terminatingReason \+
      currentHandlerInformed := true
            if state = 'TERMINATING' and:
            [(terminatingReason = 'completed')
            and: [currentHandlerInformed not]]
end-parallel
```

Thus, the state is only set to 'IDLE' if `currentHandlerInformed` is true. In turn, `currentHandlerInformed` is only set to true if the state is 'TERMINATING'. This implies that the state is only set to 'IDLE' if it is currently in the 'TERMINATING' state. This concludes the informal correctness arguments for B2.1.

Thus, due to the **data encapsulation** provided by the SLOOP method, the instance variables of an object can only be modified by that object itself or by methods provided by that object to its clients. If a method is provided that allows clients to set the value of the instance variable under consideration (`state` in the above example) to the value of interest ('IDLE' in the above example), then all the statements of the program need to be checked to determine whether this method is being invoked by any of them. However, if no such method exists (as in the above example), then the statements that need to be examined are reduced to those appearing in the class definition itself. The procedures used during correctness reasoning in the SLOOP method therefore take advantage of the **structuring** and **data encapsulation** capabilities of object-orientation.

Part B2.2:

In this part it needs to be shown that the `ServiceProviderSimulator` instances are the only objects that invoke the `terminate:` method of the `Connection` class. Note that up until now the correctness arguments have all been presented from first principles. However, one of the advantages of an object-oriented method such as SLOOP is the fact that correctness arguments can also be **reused**. This is first illustrated in this part and then applied in all the remaining correctness arguments.

The only method that causes the connection state to change to 'TERMINATING' is the `terminate:` method of the `Connection` class. The only class that invokes the `terminate:` method of the `Connection` class at this level of refinement is the `ServiceProviderSimulator` class[14]. This is determined by inspecting the correctness properties of all the classes and their methods. It is found that the following property of the `p_generateEvent` method of the `ServiceProviderSimulator` class refers to the `terminate:` method:

```
"precedence property of the p_generateEvent method"
generatingEvent ∧ serviceRequest notNil ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧
      serviceRequest isNil ∧ ¬generatingEvent
                        "DP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the connection currently associated with the service request is terminated and that the service provider simulator becomes available to service a new service request."

[14] The ServiceProviderSimulator class is specified in Appendix B, Section B.13.

The `postconditions:withArguments:` construct was first discussed in Chapter 4, Section 4.3.4.2. To recapitulate: it is used when a method sends a message to another object and the postconditions of the method being executed by the other object have significance in the correctness properties of the sending method. Thus, in the case of the *DP1-01* property of the `ServiceProviderSimulator`, the postconditions of the `terminate:` method of the `Connection` class will hold in addition to the postconditions that will hold as a result of assignments to attributes of the `ServiceProviderSimulator` instance. These postconditions will hold if `'completed'` is used as the argument of the message.

The introduction of such a construct has several advantages. It **prevents** the problem of **inconsistency** which could arise if the same postconditions were specified in multiple places and it also **highlights the use of another method** in the property specification. This is particularly useful in correctness arguments such as the current one, where one is trying to identify all the methods that are invoking the `terminate:` method.

Note that it is not necessary to verify at this point that the `p_generateEvent` method indeed invokes the `terminate:` method. The properties of the `ServiceProviderSimulator` class are reused without proving them from first principles. The correctness of the methods of the `ServiceProviderSimulator` class are shown at the time when the correctness arguments for the `ServiceProviderSimulator` class itself are presented. Thereafter it can be assumed that the behaviour of this class is as specified by its correctness properties.

Part B2.3:

It now needs to be shown that the `ServiceProviderSimulator` instance only invokes the `terminate:` method if it has a service request assigned to it. This follows directly from the *DP1-01* property of the `p_generateEvent` method of the `ServiceProviderSimulator` class:

```
"precedence property of the p_generateEvent method"
generatingEvent ∧ serviceRequest notNil ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil ∧
      ¬generatingEvent  "DP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the connection currently associated with the service request is terminated and that the service provider simulator becomes available to service a new service request."

Part B2.4:

It has now been argued that a connection is only terminated by the service provider simulator and then only if it has a service request associated with it. It must be shown next that after the service request has been assigned to a service provider simulator, it is no longer present in the `inputQ`. The following precedence properties of the `CC_SimulationActivation` and `CC_Activation` classes respectively are relevant.

```
<∀ aServiceRequest where
      < ∃ aServiceProviderSimulator where
      spAgentContainer includes: aServiceProviderSimulator::
            aServiceProviderSimulator serviceRequest = aServiceRequest
      ::
         <∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
         > precedes
            aServiceProviderSimulator serviceRequest = aServiceRequest
      >
>                                    "AP2-01 (CC_SimulationActivation)"
```

"A service request is assigned to an element of the service provider container only if the former has been enqueued in a service queue and has remained in the queue until it was assigned to the service provider container element."

```
<∀ aServiceRequest where
      < ∃ aServiceQueue ::
            aServiceQueue includes: aServiceRequest
      ::
         inputQ includes: aServiceRequest
         precedes
            aServiceQueue includes: aServiceRequest
      >
>                                                    "AP2-02 (CC_Activation)"
```

"A service request is allocated to a service queue only if the service request has been enqueued in the inputQ and has remained in the latter until it was allocated to the service queue."

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest ensures
      ¬(inputQ includes: aServiceRequest) ∧
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest)
      >
>                                                    "AP1-05 (CC_Activation)"
```

"A service request remains in the inputQ until it is assigned to a service queue."

In the above properties `aServiceQueue` is quantified over all service queues associated with `ServiceCategory` instances in the `scContainer`. This quantification is not shown in order to make the property specifications less cluttered.

The argument is as follows: Property *AP2-01* specifies that a service request can only be assigned to a service provider simulator if it comes from a service queue. In turn, a service request can only be allocated to a service queue if it comes from the `inputQ` (property *AP2-02*). Furthermore, when a service request is allocated to a service queue, it is removed from the `inputQ` (property *AP1-05*). Since there is no statement which adds the service request to the `inputQ` while it is in the service queue or while it is assigned to a service provider simulator, it follows that the service request is not present in the `inputQ` while it is assigned to a service provider simulator.

There is no statement which adds a service request to the `inputQ` once it has been removed, except if the connection has been terminated and has reached the 'IDLE' state (from Part B1). Thus, a service request that is an element of the `inputQ` is no longer present in the `inputQ` by the time the state of the associated connection changes to 'IDLE'. This concludes Part B2 of the correctness arguments.

Thus, the above example has illustrated the following:

- **Location counters are not considered** during correctness reasoning.
- It is also **not necessary** to be concerned about the **allocation of statements to processors**. At the design level, correctness reasoning is in terms of the **parallel statements**, each of which executes atomically.
- **Data encapsulation** provides a mechanism to reduce the effort required during correctness reasoning.
- The **reuse** of correctness properties significantly reduces the correctness reasoning effort.

7.3.1.3 Using a global invariant property to discuss the responsibility of the client object regarding preconditions in correctness properties

This section focuses on the **responsibilities of the client object** in ensuring that **preconditions are satisfied** when a method is invoked. In Chapter 5, section 5.4.1.3, the following property was specified in order to ensure that a service request does not get overwritten by another one before its processing has been completed:

AS3-09. A service provider / service provider simulator services a single service request at a time.

Specifying the correctness property formally

In the `CC_SimulationActivation` class, this property is specified as follows:

```
<∀ aServiceProviderSimulator where
      spContainer includes: aServiceProviderSimulator ::
      <∀ aServiceCategory where
      scContainer includes: aServiceCategory ::
      invariant aServiceProviderSimulator serviceRequest notNil ⇒
            ¬aServiceProviderSimulator canAcceptNextSR:
            (aServiceCategory serviceQCategory)
      >
>                                          "AS3-09 (CC_SimulationActivation)"
      "A service provider simulator services a single service request at a time."
```

Thus, once a service request is allocated to a service provider simulator, no other service request can be allocated to that service provider simulator until the latter has completed servicing the first service request. It is therefore not possible to overwrite the first service request.

The strategy to be followed for the correctness arguments

Since this is an invariant, it has to be shown that the property holds initially (part A), as well as after the execution of each parallel statement (part B). There are two aspects to the arguments in part B. Firstly, it needs to be shown that the `ServiceProviderSimulator` instance does not accept a new service request if another service request is already assigned to it (Part B1). If it is found that the `ServiceProviderSimulator` class achieves this via the preconditions of the method which accepts a new service request for processing, then it also has to be shown that those preconditions are indeed met whenever that method is invoked by a client (Part B2).

This is because the above correctness property is not only required to hold for the `ServiceProviderSimulator` class, but also for the system as a whole. (Property *AS3-09* is a correctness property of the composite `CC_SimulationActivation` class.)

Since the behaviour of the target object is undefined if the preconditions are not met, it is imperative to check that the client only invokes the method under the correct circumstances. This is called the "demanding" design approach in [Meye97], where the target object expects its methods to be invoked only if their respective preconditions are met and where the methods of the target object do not contain code to take some action if the preconditions are not met.

The motivation for a "demanding" design approach is discussed at length in [Meye97]. It is argued that the client object is best equipped to deal with the cases where the preconditions cannot be met (often the target object cannot do anything more constructively than print an error message). The example in this section demonstrates the impact on correctness reasoning when this approach is adopted.

The correctness arguments

Part A:
Immediately after a `ServiceProviderSimulator` instance has been created and initialized, it does not have any service request assigned to it, as can been seen in the SLOOP specification of the `ServiceProviderSimulator` class in Appendix B, Section B.13. Property *AS3-09 (CC_SimulationActivation)* is therefore satisfied initially.

Part B:
It now has to be shown that the property holds after the execution of any parallel statement of the program. As discussed in the previous section, the magnitude of the task is reduced by the fact that only those statements that are relevant to this property need to be considered.

Part B1:
It is evident from the total correctness property of the `processServiceRequest:` method of the `ServiceProviderSimulator` class that the `ServiceProviderSimulator` class ensures that property *AS3-09* is not violated. (The parameter that is passed in the `processServiceRequest:` message is called `aServiceRequest` and it refers to the new service request.)

```
"Total correctness property of the processServiceRequest:
method"
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Thus, the precondition for accepting a new service request is that the `canAcceptNextSR:` method should return true. From the total correctness property of the latter it is clear that the `canAcceptNextSR:` method only returns true if no service request is assigned to that `ServiceProviderSimulator` instance (and the next service category to be serviced matches the one passed as parameter), as can be seen below:

```
"total correctness property of the canAcceptNextSR: method"
true results-in
      methodReturnValue = ((requestingServiceCategory =
      categoriesServed at: (categoryIndex + 1)) ∧
      (serviceRequest isNil))
                          "DL1-04 (ServiceProviderSimulator)"
```

A `ServiceProviderSimulator` instance therefore only accepts a new service request if it is not currently processing another one.

Part B2:
It is the responsibility of the client to ensure that `canAcceptNextSR:` returns true prior to invoking the `processServiceRequest:` method. If the client does not ensure that the precondition of the total correctness property (*DL1-06*) of `processServiceRequest:` holds, the `ServiceProviderSimulator` instance does not have any obligations regarding the correctness properties of the class and its methods. In the call centre example, the `ServiceProviderSimulator` class overwrites the existing service request in the

processServiceRequest: method if the preconditions are not met, as is evident from the statements of this method:

```
"The processServiceRequest: method:
sequential
newEventRequired := true \+
serviceRequest := aServiceRequest \+
categoryIndex := (categoryIndex + 1) \\ nrOfCategoriesServed
end-sequential
```

In the correctness arguments of property *AS3-09 (CC_SimulationActivation)* it is therefore necessary to show that the precondition will always hold when the processServiceRequest: method is invoked. The only statement which invokes the latter is the assignToSP: method of the ServiceCategory[15] class.

The following total correctness property applies to the assignToSP: method:

```
sr notNil ∧ availableServiceProvider notNil results-in
      methodReturnValue = self ∧
      availableServiceProvider
            postconditions:(#processServiceRequest:)
            withArguments: #(sr)  ∧
      sr postconditions: (#serviceProvider:)
            withArguments: #(availableServiceProvider)
                                          "DL1-10(ServiceCategory)"
```

where availableServiceProvider is defined in a macro as:

```
availableServiceProvider ≡
      spSubset detect: [:each | each canAcceptNextSR:
      serviceQCategory]
```

Thus, the ServiceCategory instance assigns the service request denoted by sr to the first service provider simulator in spSubset which can accept another service request. (The spSubset instance variable of the ServiceCategory class represents the set of service provider simulators that have the capability to process service requests that are enqueued in the serviceQ of this ServiceCategory instance.)

The processServiceRequest: message is therefore only sent to the ServiceProviderSimulator instance if the canAcceptNextSR: message to that instance has returned true. This concludes the informal correctness arguments regarding property *AS3-09 (CC_SimulationActivation)*.

This section has shown how a class can restrict its responsibilities regarding the preservation of a property by specifying the appropriate preconditions for its methods. It is then the **responsibility** of the **client** of these methods to ensure that the relevant preconditions are met when these methods are invoked.

7.3.1.4 Using an unless property to demonstrate how the correctness properties specified for the class itself as well as those inherited from its parent class are applied in correctness arguments

It is important to understand the effect of **inheritance** on correctness properties. There are three distinct cases:

- A correctness property may be **reused as is** in the descendant.
- A correctness property may be **added** in the descendant.
- A correctness property of the descendant may **override** one in its ancestor.

[15] The ServiceCategory class is specified in Appendix B, Section B.10.

The first two cases are covered in the first example described in this section. Subsequently, another example is given which demonstrates how a correctness property can be overridden by a descendant.

The following is one of the **unless** properties of the `ServiceProviderSimulator` class :

```
serviceRequest isNil ∧ ¬newEventRequired unless
      serviceRequest notNil ∧ newEventRequired
                              "AS4-01 (ServiceProviderSimulator)"
```

"When a new service request is assigned to the service provider simulator then a new service provider simulator event is required."

Property *AS4-01* specifies that when the value of the `serviceRequest` instance variable changes from nil to not nil, then the value of the `newEventRequired` instance variable changes from false to true. This is an example of a correctness property that is **added** in a descendant. The `EventSimulator` class, which is the parent of the `ServiceProviderSimulator` class[16], does not contain this property at all. However, it will now be shown how some of the properties defined for the parent class are **reused as is** in the correctness arguments for property *AS4-01(ServiceProviderSimulator)*.

The strategy to be followed for the correctness arguments

Since the `ServiceProviderSimulator` instance does not provide any methods to clients to modify the `serviceRequest` and `newEventRequired` instance variables, it is merely necessary to check the statements of the `ServiceProviderSimulator` class and its ancestors in order to verify that this property is not violated.

First of all it is shown in **part A** that when the value of `serviceRequest` changes from nil to not nil, then `newEventRequired` is set to true. Thereafter it is shown in **part B** that `serviceRequest` is only set to a non-nil value when the value of `newEventRequired` changes from **false** to true, i.e. `newEventRequired` is not already true when `serviceRequest` is set to a non-nil value.

Part B has two subsections:
B1) Firstly it is shown that `newEventRequired` is set to false and `serviceRequest` is set to nil upon initialization. Thus, when `processServiceRequest:` is executed the first time, `newEventRequired` is false, since no other method sets this variable to true. (It is in the `processServiceRequest:` method where `serviceRequest` is set to a non-nil value and where `newEventRequired` is set to false.)

B2) It must then be shown that whenever `processServiceRequest:` is invoked after that, `newEventRequired` has the value false. When `processServiceRequest:` is executed, `newEventRequired` is set to true and `serviceRequest` is set to a non-nil value. Since one of the preconditions of the `processServiceRequest:` method is that `serviceRequest` should be nil, it means that the `processServiceRequest:` method can only be executed again once `serviceRequest` has been set to nil. Note that it is not a precondition of this method that `newEventRequired` should be false. In order to be sure that `processServiceRequest:` is not invoked while `newEventRequired` is still true, it therefore has to be shown that `serviceRequest` will remain not nil for at least as long as `newEventRequired` is true.

[16] The ServiceProviderSimulator class is specified in Appendix B, Section B.13 and its parent class, the EventSimulator class, is specified in Appendix B, Section B.5.

Once `newEventRequired` is true, a timer will eventually be started, at which point `newEventRequired` is set to false. The `serviceRequest` variable is only set to nil once this timer has expired. This means that `newEventRequired` is set to false before `serviceRequest` is set to nil. Since `processServiceRequest:` is the only method which sets `newEventRequired` to true, it follows that `newEventRequired` will always be false when `serviceRequest` is nil, i.e. when `processServiceRequest:` is invoked. The sequence of events as dictated by the preconditions of the above methods is shown below in tabular format. (The way in which these preconditions determine the sequence of events is referred to as synchronisation constraints, a topic which was discussed in Chapter 3, Section 3.2.2.3.)

Step	Method executed	Variables modified, methods invoked
1	`processServiceRequest:`	`newEventRequired := true` `serviceRequest := aServiceRequest`
2	`p_simulate: timeoutEventsIn:`	`newEventRequired := false` `startRandomTimer:withMaximum`
3	`p_simulate:timeoutEventsIn:`	`generatingEvent := true` `resetTimerExpired:`
4	`p_generateEvent`	`serviceRequest := nil` `generatingEvent := false`

Table 7-1. Sequence of events involving the `newEventRequired` and `serviceRequest` instance variables.

The correctness arguments

PartA:
It needs to be shown that when the value of `serviceRequest` is set to not nil, then `newEventRequired` is set to true. There is only one method which sets the `serviceRequest` instance variable to a non-nil value, viz. `processServiceRequest:`. The total correctness property of this method specifies that when the value of `serviceRequest` changes from nil to not nil, then `newEventRequired` is set to true:

```
"Total correctness property of the processServiceRequest:
method"
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Property *DL1-06* therefore guarantees that when the value of `serviceRequest` changes from nil to not nil, then `newEventRequired` is set to true.

Part B:
It must now be shown that `newEventRequired` will change from **false** to true when the value of `serviceRequest` is set to a non-nil value.

The total correctness property of the `processServiceRequest:` method guarantees that `newEventRequired` is set to **true** when the method is **executed**, as was shown above. It therefore remains to be shown that `newEventRequired` will always be **false** when the `processServiceRequest:` method is **invoked**.

Part B1:
From the total correctness property of the `initialize` method of `EventSimulator`, the parent class, it is clear that `newEventRequired` is false initially.

```
"Total correctness property of the initialize method"
true results-in methodReturnValue = self ∧
rand notNil ∧ newEventRequired = false ∧
currentRandomTimeoutValue = 1 ∧
generatingEvent = false ∧
timerOutstanding = false                    "DL1-01 (EventSimulator)"
```

The `moreInit:using:` method of the `ServiceProviderSimulator` class sets `serviceRequest` to nil initially as can be seen from the total correctness property of that method.

```
"Total correctness property of the moreInit:using: method"
true results-in methodReturnValue = self ∧
      serviceRequest isNil ∧
      aConfiguration postconditions: (#assignSPCategory) ∧
      serviceProviderCategory notNil ∧
      categoriesServed notNil ∧
      self postconditions: (#registerServiceProvider: using:)
            withArguments: #(scContainer aConfiguration)
                        "DL1-01 (ServiceProviderSimulator)"
```

Since both these methods are executed when the `ServiceProviderSimulator` instance is created, `newEventRequired` is false and the value of `serviceRequest` is nil when the `processServiceRequest:` method is invoked the first time. This is because `processServiceRequest:` is the only method which sets `newEventRequired` to true and `serviceRequest` to not nil. It now remains to be shown that `newEventRequired` will be false whenever `processServiceRequest:` is executed thereafter.

Part B2:
The safe liveness property of the `p_simulate:timeoutEventsIn:` method of the `EventSimulator` class (i.e. the **parent** class) ensures that `newEventRequired` will eventually become false, as can be seen below:

```
newEventRequired ensures
      self postconditions: (#startRandomTimer:withMaximum:)
      withArguments:
      #(aTimerServices (aTimerServices maximumTimeout))
      ∧ ¬newEventRequired              "DP1-01(EventSimulator)"
```

"When newEventRequired is true, it ensures that a simulation timer is started and newEventRequired becomes false."

Since the `processServiceRequest:` method is the only one that sets `newEventRequired` to true, `newEventRequired` will remain false until `processServiceRequest:` is executed again. The latter is not executed while `serviceRequest` has a non-nil value. One therefore has to show next that `serviceRequest` will only be set to nil once `newEventRequired` has already been set to false.

Property *DP1-01(ServiceProviderSimulator)* describes the conditions under which `serviceRequest` is set to nil. It specifies that it will only happen if `generatingEvent` is true, as can be seen below:

```
generatingEvent ∧ serviceRequest notNil ensures
        (serviceRequest connection) postconditions: (#terminate:)
        withArguments: #('completed') ∧
        serviceRequest isNil ∧ ¬generatingEvent
                                "DP1-01 (ServiceProviderSimulator)"
```

"If a service provider simulator has to generate an event, it ensures that the connection currently associated with the service request is terminated and that the service provider simulator becomes available to service a new service request."

As is evident from property *DP1-02* of the `EventSimulator` **parent** class, `generatingEvent` is only set to true if a timer started by `self` expires. In this case `self` is the current `ServiceProviderSimulator` instance.

```
self timerExpired: timerEventQ ensures
        generatingEvent ∧
        self postconditions: (#resetTimerExpired:)
        withArguments: #(timerEventQ) "DP1-02(EventSimulator)"
```

"When a simulation timer expires, it ensures that generatingEvent becomes true."

The total correctness property of the `timerExpired:` method of the `EventSimulator` **parent** class specifies that it will only return true if the `timerEventQ` contains an element which has `self` as the `timeoutRequestor`, i.e. the current `ServiceProviderSimulator` instance requested the timer earlier on.

```
"Total correctness property of the timerExpired: method"
true results-in methodReturnValue =
        (timerEventQ detect: [:each |
        each timeoutRequestor == self ]
        ifNone: [nil]) notNil          "DL1-03(EventSimulator)"
```

Thus, `generatingEvent` is **only set to true upon the expiry of a timer** started by the current `ServiceProviderSimulator` instance. Property *DP1-01* of the `EventSimulator` **parent** class guarantees that `newEventRequired` is set to false when an `EventSimulator` subclass requests a timer to be started, as can be seen below.

```
newEventRequired ensures
        self postconditions: (#startRandomTimer:withMaximum:)
        withArguments:
        #(aTimerServices (aTimerServices maximumTimeout))
        ∧ ¬newEventRequired            "DP1-01(EventSimulator)"
```

"When newEventRequired is true, it ensures that a simulation timer is started and newEventRequired becomes false."

Thus, the expiry of a timer has to be preceded by the setting of that timer, at which point `newEventRequired` is set to false. The value of `serviceRequest` can therefore only be set to not nil once `newEventRequired` has been set to false.

The correctness arguments for part B are summarised as follows:
In order to show that `newEventRequired` is always false when `processServiceRequest:` is executed, it was first shown that `newEventRequired` is set to false and `serviceRequest` is set to nil upon initialization. Thus, when `processServiceRequest:` is executed the first time, `newEventRequired` is false, since no other method sets this variable to true.

Thereafter, the following occurs: When `processServiceRequest:` is executed, `newEventRequired` is set to true and `serviceRequest` is set to a non-nil value. Eventually, a timer is started, at which point `newEventRequired` is set to false. Once the timer expires, `generatingEvent` is set to true. Only once `generatingEvent` is true, can

`serviceRequest` be set to nil. Since `generatingEvent` is only set to true upon expiry of a timer, it follows that once `processServiceRequest:` has been executed, `newEventRequired` is always set to false before `serviceRequest` is set to nil. This concludes the correctness arguments for property *AS4-01*.

In order to reason about the correctness of the behaviour of the `ServiceProviderSimulator` class, it is necessary to take the correctness properties of its parent class, `EventSimulator`, into account. The above example has therefore demonstrated how correctness properties that are **inherited** from a **parent** class are **reused** in the correctness arguments of properties of a **descendant** class.

Overriding a property of an ancestor

The `CC_SimulationActivation` class provides an example of where a correctness property **overrides** a property of an ancestor. The `initCommsAgent` method of the `CC_Activation` class has the following total correctness property:

```
"Total correctness property of the initCommsAgent method"
true results-in methodReturnValue notNil
                                        "DL1-05 (CC_Activation)"
```

The `CC_SimulationActivation` subclass **overrides** the `initCommsAgent` method and the above property is **specialized** in the subclass in the following way:

```
"Total correctness property of the initCommsAgent method"
true results-in methodReturnValue notNil ∧
      CC_SimulationInterfacesPkg:::CommsProviderSimulator
      postconditions: (#startSimulation)
                                        "DL1-05 (CC_Activation)"
```

Thus, the `CC_Activation` class merely specifies that the method will return a non-nil value, but it does not specify which class should be instantiated. That is left up to the subclass.

The overriding of a property is indicated syntactically by repeating the number of the property from the ancestor and including the name of the ancestor in brackets. This is similar to the way in which a method in a descendant overrides a method with the same name in the ancestor. (All properties that are not inherited are assigned identifiers that are unique within that class.)

In the above example the preconditions are left unchanged, while the **postconditions** are **strengthened** in the **subclass**. It is important that the designer should take into account that **preconditions** may **not** be **strengthened** and **postconditions** may **not** be **weakened** when correctness properties are overridden in a subclass. This is to ensure that a class could be replaced with its subclass while guaranteeing that all the properties that used to hold for the parent class will still hold when the subclass is used instead [Meye97].

In order to ensure that preconditions can be weakened and postconditions can be strengthened, care should be taken during the specification of properties to avoid overspecification. For example, in the `TimerServices`[17] class, the size of the `timeoutCollection` array is dependent on the maximum timeout value (it is equal to `maximumTimeout + 2`). It is important that all other properties that depend on the size of the `timeoutCollection` array should refer to '`timeoutCollection size`', rather than '`maximumTimeout + 2`', since the size of the `timeoutCollection` array might be calculated differently in subclasses of

[17] The TimerServices class is described in Appendix B, Section B.11. Details about its design and its properties are also given in Section 7.3.2.2.

`TimerServices`. For such subclasses it should not be necessary to modify the properties of the methods that use the size of the `timeoutCollection` array.

For example, the clean behaviour property of the `start:id:for:` method should not have to change if the size of the `timeoutCollection` array is calculated differently.

```
invariant    1 ≤ writeIndex ∧ writeIndex ≤ timeoutCollection size
                                        "DS3-02(TimerServices)"
```

The above examples have demonstrated how a correctness property first defined in a parent class can be **reused as is** in a descendant class. It has also been shown how correctness properties can be **added** in descendant class. The third type of reuse allowed for the **specialization** of a correctness property in a descendant class, provided the modifications adhere to the rule given earlier in this section. This concludes the discussion about the impact of **inheritance** on correctness arguments.

7.3.2 Liveness properties

This section deals with three different types of liveness properties, viz. total correctness, intermittent assertions and responsiveness properties. The correctness arguments for examples of these properties are used as a vehicle to discuss various aspects of the correctness reasoning in the SLOOP method. The first example emphasizes the atomicity of sequential methods and how this characteristic can be used in correctness properties. The intermittent assertion example shows how the computational model is used to reason about progress and the responsiveness example demonstrates the importance of showing that preconditions will eventually hold when correctness properties are being reused in correctness arguments of liveness properties.

7.3.2.1 Showing why the postconditions of a total correctness property can be used in an ensures relation

The sequential methods in a SLOOP program are either executed as part of the sequential statements in the *activation-section* or they are invoked from within parallel statements. Since each parallel statement is executed **atomically** (i.e. **statement interleaving** takes place at the level of **parallel** statements), concurrency does not impact on sequential statements embedded in parallel statements.

The safe liveness property of the `p_doWrapUp` method of the `Connection` class demonstrates how the postconditions of a sequential method can be used in an **ensures** relation. Recall that the **ensures** relation only applies if the postcondition of the relation is reached via the execution of a **single** SLOOP statement. Although the `reset` method of the `ServiceRequest` class that is invoked by the `p_doWrapUp` method of the `Connection` class contains multiple sequential statements, they are embedded in the single parallel statement of the `p_doWrapUp` method, as shown below:

```
"p_doWrapUp method of the Connection class"
message pattern p_doWrapUp
method properties
"Safe liveness"
currentHandlerInformed ensures
state = 'IDLE' ∧ serviceRequest postconditions: (#reset) ∧
¬currentHandlerInformed                    "DP1-02(Connection)"
parallel
state := 'IDLE' \+
serviceRequest reset \+
currentHandlerInformed := false
      if currentHandlerInformed
end-parallel
```

```
"reset method of the ServiceRequest class"
message pattern reset
method properties
"Total correctness"
true results-in methodReturnValue = self ∧ serviceQ isNil ∧
      serviceRequestCategory isNil ∧ serviceProvider isNil ∧
      categorisationData isNil       "DL1-12(ServiceRequest)"
sequential
serviceQ := nil
[] serviceRequestCategory := nil
[] serviceProvider := nil
[] categorisationData := nil
end-sequential
```

Thus, despite the fact that the `reset` method contains four separate SLOOP statements, each one achieving part of the postcondition, the client of a sequential method views the execution of such a method as an atomic event. The postconditions of a sequential method may therefore be used in the postconditions of an **ensures** relation. In this example it is therefore guaranteed that no other parallel statements can be executed while the `Connection` instance and its associated `ServiceRequest` instance are being reset.

The **correctness arguments** to reason about a total correctness property of a sequential method proceed as for any **conventional sequential program, but taking the semantics of SLOOP sequential statements into account**. For example, if a statement contains a *conditional-component-part-list*[18] as in the `registerServiceProvider:using:` method of the `ServiceProviderSimulator` class below, then all the *component-parts* of the list are subject to the associated condition.

The order of the statements is significant. For example, in the code fragment below it is clear that the statement setting `nrOfCategoriesServed` to zero has to be executed before the loop that follows. This is because this variable is incremented within the loop whenever the simulator is registered with a service category. Once the loop has been completed, this variable contains the number of service categories serviced by this simulator.

If another method is invoked by any of the statements, then the semantics for the invocation are as for a function call in a conventional programming language. The total correctness property of the `registerServiceProvider:using:` method below specifies the conditions that should hold when the method is entered and it also specifies the conditions that should hold when control exits from the method.

```
message pattern registerServiceProvider: scContainer
            using: aConfiguration
"Registers the ServiceProviderSimulator with the relevant
service categories"
method macros
maxCategories ≡ aConfiguration maximumServiceCategories
method properties
"Total correctness"
true results-in methodReturnValue = self ∧
      <∀aServiceCategory where
      scContainer includes: aServiceCategory ∧
      aServiceCategory servicedBy: serviceProviderCategory ::
```

[18] The *conditional-component-part-list* was defined in Chapter 4, Section 4.3.6.2.

```
        aServiceCategory postconditions: (#addSP:)
        withArguments: #(self) ∧
        categoriesServed includes:
              (aServiceCategory serviceCategory)
        > ∧
        nrOfCategoriesServed = categoriesServed size ∧
        categoryIndex ≥ 0  "DL1-05 (ServiceProviderSimulator)"
sequential
  nrOfCategoriesServed := 0
[] < [] j where 1≤j≤maxCategories ::
(scContainer at: j) addSP: self \+
nrOfCategoriesServed := nrOfCategoriesServed + 1 \+
categoriesServed addLast:((scContainer at: j) serviceCategory)
      if (scContainer at: j) servicedBy: serviceProviderCategory
>
[] categoryIndex := 0
end-sequential
```

Since there can be **no interference** while the statements of a **sequential method** are executing, it is guaranteed that once the condition of a conditional statement within the sequential method has been evaluated, its value cannot change before the rest of the statement starts executing.

The examples in this section have highlighted the way in which the execution of sequential methods should be interpreted, i.e. the method executes as an **atomic unit**. No concurrency needs to be taken into account when reasoning about the behaviour of the statements within a sequential method. As far as the computational model is concerned, a sequential method is executed as part of a **single parallel** statement. The total correctness properties of a sequential method can therefore be used in an **ensures** relation (and in the other SLOOP relations as well).

7.3.2.2 Using an intermittent assertion property to demonstrate how the repeated execution of parallel statements guarantees progress, provided the preconditions hold at some point

The aim in this section is to demonstrate how the **repeated execution of the parallel statements** selected for the program **eventually results in the postconditions** specified by the **liveness** properties, provided that their preconditions hold at some point. The example is taken from the list of intermittent assertion properties specified in Chapter 5:
AL2-01.Once a timer has been started, it will eventually expire or it will be stopped.

Specifying the correctness property formally

In order to write this property more formally, it is necessary to consider how timers are represented in the system. A brief description of the design of the `TimerServices` class was presented in Chapter 6, Sections 6.2.4 and 6.3.1, and full details are given in Appendix B, Section B.11. However, for convenience, a short summary is presented here.

The `TimerServices` instance, which handles all timer requests, creates a `TimeoutElement` instance whenever a timer is started. In order to be able to inform the requestor of the expiry of the timer, the `TimerServices` instance determines when the timer would expire and then stores the `TimeoutElement` instance in the list of timers that will expire at the calculated time. These **lists** are stored in an instance of the `Array` class, called `timeoutCollection`.

The `TimerServices` class implements this array as a **circular** array. Each position in the array represents one second. The `TimerServices` instance maintains an index into the array. This

index is called `currentTick` and is advanced every second (it is incremented modulo the size of the array). Thus, the entry in the array which will be reached x seconds from the current moment can be calculated using the value of `currentTick`, the size of the array and the value of x. Each entry in the array is an ordered collection of `TimeoutElement` instances.

When a timer expires, its associated `TimeoutElement` instance is removed from the list in `timeoutCollection` and entered into `timerEventQ`, which is checked by all timer requestors on a regular basis. Appendix B, Section B.11, contains two diagrams (Figures B-3(a) and B-3(b)) illustrating the above concepts.

The responsibility for ensuring that each timer will expire unless it is stopped, lies with the `TimerServices` class. When property *AL2-01* is specified more formally, it can therefore be written in terms of the instance variables of the `TimerServices` class:

```
<∀ aTimeoutElement where
      <∃ i where 1 ≤ i ≤ (timeoutCollection size) ::
      (timeoutCollection at: i) includes: aTimeoutElement
      > ::
      ¬aTimeoutElement timerServicesCompleted leads-to
            aTimeoutElement timerServicesCompleted
>                                          "DL2-01(TimerServices)"
```

"Once a timer has been started, i.e. it is present in one of the lists associated with timeoutCollection, the TimerServices instance will eventually complete its responsibilities regarding the timer (i.e. the timer will either be stopped or the TimerServices instance will indicate its expiry to the requestor of the timer)."

The strategy to be followed for the correctness arguments

It needs to be shown that once a `TimeoutElement` instance has been entered in one of the lists reached via the `timeoutCollection` array, the `TimeoutElement` instance will eventually be removed from the relevant list because the timer has been stopped by a client of the `TimerServices` instance or it will be removed because the timer has expired, in which case it is added to the `timerEventQ` list. Abnormal conditions are not considered at this level of abstraction, therefore the case where the timer could be stopped is ignored in this discussion. (At this level of abstraction the `TimerServices` class does not export a method to stop a timer.) The strategy for the correctness arguments is therefore as follows:

First of all it needs to be shown that the index which identifies the list of expired timers (the `readIndex`) will eventually reach the list containing the specified `TimeoutElement` instance (part A). It then needs to be shown that once that happens, the `TimeoutElement` instance will eventually be removed from the list and added to the `timerEventQ` (part B). Since progress is achieved via the infinitely often execution of parallel statements, the obvious place to start is at the parallel method defined for the `TimerServices` class, namely `p_runTimer:`. This method contains three parallel statements, as seen below:

```
parallel
   currentTime := SmalltalkLibPkg:::Time now asSeconds      "S1"
[] lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 and: [currentTimeoutElement isNil] "S2"
[] timerEventQ addLast: currentTimeoutElement \+
   currentTimeoutElement updateEndTime \+
   currentTimeoutElement timerServicesCompleted: true \+
   (timeoutCollection at: readIndex) removeFirst
      if currentTimeoutElement notNil                       "S3"
end-parallel
```

The purpose of each statement is summarised here, with more detail to follow. The first statement (*S1*) updates the `currentTime` instance variable with the current time (in seconds) whenever the statement is executed. The second statement (*S2*) updates two instance variables, viz. `lastTime` and `currentTick`, whenever at least one second has expired and all the `TimeoutElement` instances in the list identified by the current `readIndex` have been removed. The `lastTime` instance variable records the last time when `currentTick` was updated and is used to determine whether one second has already expired since the last update. Two *macro-variables*, viz. `difference` and `currentTimeoutElement` are used in this statement. They will described in more detail later. The last statement (*S3*) is used to remove the first `TimeoutElement` instance from the list identified by the `readIndex` and to add it to `timerEventQ`. The correctness arguments for property *DL2-01 (TimerServices)* will refer to the above statements repeatedly.

The correctness arguments

Part A:
The `readIndex` *macro-variable* is defined as follows in the `p_runTimer:` method of the `TimerServices` class (the one has to be added because SLOOP array indices start at one, not zero):

```
readIndex ≡ currentTick + 1
```

Since the `readIndex` is defined in terms of `currentTick`, it needs to be shown that `currentTick` will eventually be advanced from its current position, regardless of where that position is. The following statement (*S2*) in the `p_runTimer:` method advances the position of `currentTick`:

```
[] lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 and: [currentTimeoutElement isNil]   "S2"
```

The value of `currentTick` is incremented modulo the size of the `timeoutCollection` array, which implies the following:

```
invariant 0 ≤ currentTick ∧ currentTick ≤ (timeoutCollection size) - 1.
```

The advancement of `currentTick` depends on two conditions, involving the `difference` and `currentTimeoutElement` *macro-variables* respectively. These variables are defined as follows:

```
difference ≡ currentTime - lastTime
                  if (currentTime - lastTime) ≥ 0 ~
               currentTime + (86400 - lastTime)
                  if (currentTime - lastTime) < 0
[] currentTimeoutElement ≡
   (timeoutCollection at: readIndex) first
      if (timeoutCollection at: readIndex) isEmpty not ~
   nil
      if (timeoutCollection at: readIndex) isEmpty
```

First of all it needs to be shown that `difference` will eventually be greater than or equal to one. As can be seen from the above, `difference` is used to calculate the elapsed time since `currentTick` was last updated. It takes care of the rollover at midnight. The `currentTime` instance variable is updated via the following parallel statement of the `p_runTimer:` method.

```
currentTime := SmalltalkLibPkg:::Time now asSeconds      "S1"
```

Since statement *S1* must be executed **infinitely often**, it follows that `currentTime` will be updated infinitely often and it will therefore **eventually** result in `difference` having a value greater than or equal to one.

The second condition that has to be satisfied before `currentTick` and `lastTime` will be updated is that `currentTimeoutElement` should be nil. That happens only if the list of `TimeoutElement` instances is empty, as is evident from the *macro-definition* above. This will **eventually** become true as a result of the **infinitely often** execution of the third parallel statement of the `p_runTimer:` method. Each time that statement executes, it removes the first element of the list identified by `readIndex`, provided the list is not empty. Furthermore, the value of `readIndex` cannot change (being defined in terms of `currentTick`) until the condition for the execution of statement *S3* ceases to hold.

Thus, both conditions of statement *S2* of the `p_runTimer:` method will eventually become true, the first as a result of the infinitely often execution of statement *S1* and the second as a result of the infinitely often execution of statement *S3* of that method. Since statement *S2* is also executed infinitely often, it will eventually be executed when both conditions are true, in which case `lastTime` and `currentTick` will be updated. Thus, whatever the current value of `currentTick`, it will eventually be incremented modulo the size of the `timeoutCollection` array. Since `readIndex` is defined in terms of `currentTick`, `readIndex` will eventually identify the next list of `TimeoutElement` instances. Thus, regardless of the index of the list containing the `TimeoutElement` instance specified in property *DL2-01 (TimerServices)*, `readIndex` will eventually be equal to that index. This concludes the correctness arguments for Part A.

Part B:
It now remains to be shown that once the `readIndex` identifies a particular list of `TimeoutElement` instances, then all of those instances will eventually be removed and added to the `timerEventQ`. This follows vacuously from the infinitely often execution of statement *S3* of the `p_runTimer:` method. The presence of a `TimeoutElement` instance in the `timerEventQ` indicates to the corresponding timer requestor that the timer has expired. This concludes the correctness arguments of property *DL2-01 (TimerServices)*.

This section has focussed on the role of **parallel** statements in the correctness arguments for **progress** properties. Note that all parallel statements are executed repeatedly and in any order. If the effect of the statement is conditional, the execution of a statement may not have an effect each time it is executed.

The execution scenario as depicted in Table 7-2 shows one possible execution sequence (in this case statement *S1* is executed more often than the other two). The first statement of the `p_runTimer:` method always has an effect (it is not a conditional statement). The second statement in that method only has an effect if at least one second has expired since `currentTick` has been updated and if the list identified by `readIndex` is empty. This might not happen each time the statement is executed, as seen in the Table 7-2, where it is only updated when it is executed for the fourth time.

Statement executed	Effect
S2	`difference < 1` and there are still two elements in the list identified by `readIndex`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S3	One element is removed from the list identified by `readIndex` and added to `timerEventQ`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S2	`difference < 1` and there is still one element in the list identified by `readIndex`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S3	One element is removed from the list identified by `readIndex` and added to `timerEventQ`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S2	There are no elements in the list identified by `readIndex`, but `difference < 1`.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S3	No action is taken.
S1	`currentTime` receives the latest value of the number of seconds since midnight.
S2	`difference is now ≥ 1` and the list identified by `readIndex` is empty, therefore `currentTick` is advanced.

Table 7-2. Parallel statement execution scenario.

In this section the role of the computational model in the correctness arguments for progress properties was described. The purpose was to show how the infinitely often execution of the parallel statements eventually results in the desired outcome. This is relatively simple if all the parallel statements are unconditional. If some are conditional, however, it is also necessary that the relevant conditions should eventually become true.

In the above example it had to be shown that the conditions of parallel statement (*S2*) would eventually become true. Since the first condition depended on the infinitely often execution of parallel statement *S1*, an **unconditional** parallel statement in the `p_runTimer:` method, it was trivial to show that it would eventually become true. The second condition depended on the infinitely often execution of statement *S3*. Although statement *S3* is a conditional statement, the condition which will prevent it from executing (i.e. when there are no more elements left in the list at the `readIndex` position) is exactly the condition that is required to facilitate the execution of statement *S2*. Thus, as long as there are elements in the list at that `readIndex` position, statement *S3* will execute and remove the first element whenever it is selected for execution (which is infinitely often). Eventually this list will become empty (since no elements can be added to the list at `readIndex`[19]), thereby satisfying the second condition of statement *S2*.

[19] This is guaranteed by invariant DS3-03 (TimerServices), which is as follows:
invariant writeIndex ~= readIndex

This section has described the correctness arguments for a liveness property from first principles. In the next section it is shown how the results of other correctness properties can be reused in the correctness arguments of a liveness property. The importance of showing that the preconditions will eventually hold is also described. This is analogous to the importance of showing that the conditions of conditional parallel statements will eventually hold in the above example.

7.3.2.3 Using a responsiveness property to demonstrate the importance of showing that the preconditions will eventually hold when reusing other correctness properties in the correctness arguments of a liveness property

In this section the focus is on the **reuse** of correctness properties and the **role of preconditions** in correctness arguments for **liveness** properties. It also shows how the concept of **eventuality chains** is applied in informal liveness property proofs.

In the call centre example, the responsiveness property is a very important one, since it is the property which ensures that the service user experiences the desired effect, i.e. it ensures that a service request is eventually serviced once the user has connected to the call centre. The SLOOP specification of the call centre in Appendix B is used as the basis for the discussions in this section. The level of abstraction of that specification assumes that service users will not abort service requests and the call centre will only receive service requests that belong to categories supported by the call centre. Service providers may be idle or busy, but not completely unavailable. The responsiveness property specified for the call centre system in Chapter 5, Section 5.4.2.3, is given below for the level of abstraction used in Appendix B.

AL3-01. Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of the service provider container.

Specifying the correctness property formally

In order to specify the above property more formally, one needs to determine what predicate holds for the service request associated with the connection immediately after a service user has connected to the call centre. In the `CallCentreSimulation` program, a service user trying to establish a connection is simulated by setting the `generatingEvent` instance variable of the `CommsProviderSimulator` class to true. If an idle `Connection` instance is available, it is assigned to the service user and the associated service request is entered into the `inputQ`. This is evident from the safe liveness property of the `p_generateEvent:target:` method of the `CommsProviderSimulator` class (the `newEventRequired` instance variable is set to true in order to trigger another simulated connection request from a service user after a random timeout):

```
"Safe liveness"
generatingEvent ∧ idleConnection notNil ∧
¬(inputQ includes:(idleConnection serviceRequest)) ensures
      ¬generatingEvent ∧ newEventRequired ∧
      inputQ last = (idleConnection serviceRequest) ∧
      idleConnection postconditions: (#assign)
                                    "AP1-01(CommsProviderSimulator)"
"If an event has to be generated and the maximum number of
connections have not yet been established, the communication
provider simulator ensures that a new connection is established,
the associated service request is appended to the input queue
and a new communication provider simulator event is again
required."
```

The *macro-variable* `idleConnection` used in the above property is defined as:

```
idleConnection ≡ self getIdleConnection: userConnections
```

The `getIdleConnection:` method returns the first idle connection that can be found in the `userConnections` array, or it returns nil if there are no idle connections. This behaviour can be deduced from the total correctness properties of the `getIdleConnection:` method of the `CommsProviderSimulator` class and the `isIdle` method of the `Connection` class listed below:

```
"Total correctness property of the getIdleConnection: method"
true results-in methodReturnValue =
userConnections detect: [:each | each isIdle] ifNone: [nil]
                                        "DL1-03(CommsProviderSimulator)"

"Total correctness property of the isIdle method"
true results-in methodReturnValue = (state = 'IDLE')
                                                "DL1-05(Connection)"
```

The `serviceRequest` method of the `Connection` class returns the service request associated with the connection. The `inputQ last = (idleConnection serviceRequest)` predicate of property *AP1-01(CommsProviderSimulator)* therefore specifies that the service request associated with the first idle connection is appended to the `inputQ`. The `idleConnection postconditions: (#assign)` predicate indicates that the `assign` method is invoked on the first idle connection. That method sets the `state` of the `Connection` instance to 'CONNECTED'. The interested reader is referred to Appendix B, Section B.7 for the specification of the total correctness properties of these methods.

The fact that property *AP1-01(CommsProviderSimulator)* contains an **ensures** relation, implies that at the time when the `generatingEvent` value is changed from true to false (simulating the successful connection of the service user to the call centre), a `Connection` instance is assigned to the service user and the associated service request is entered into the `inputQ` using a **single atomic** parallel statement. Thus, when a service user connects successfully to the call centre, the service request associated with the connection is added to the `inputQ`. This provides the necessary information to specify property *AL3-01 (CC_SimulationActivation)* more formally:

```
<∀ aConnection where userConnections includes: aConnection ::
inputQueue includes: aConnection serviceRequest leads-to
      <∃ aServiceProviderSimulator where
      spAgentContainer includes: aServiceProviderSimulator ::
      aServiceProviderSimulator serviceRequest =
            aConnection serviceRequest
      >
>                                   "AL3-01 (CC_SimulationActivation)"
```

"Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of the service provider container."

Note that the above property does not specify a unique association between the service requests in the `inputQ` and the service provider simulators. For example, if there are 5 service requests in the `inputQ`, then it is not necessary to have 5 service provider simulators. One simulator would suffice. The above property merely specifies that each service request in the `inputQ` will eventually be assigned to a service provider simulator.

The strategy to be followed for the correctness arguments

The above responsiveness property can be derived in two steps:

A) by applying the transitivity rule on the **leads-to** relation of the safe liveness properties that describe the sequence of events from the time that the connection is accepted by the call centre until the associated service request is assigned to a service provider, and
B) by showing that the preconditions of each of these properties will eventually become true.

The correctness arguments

Part A:
Properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)*, first presented in Chapter 5, Section 5.4.3.1, describe the path followed by a service request through the system. It has to be shown that a service request which is present in the `inputQ` is eventually processed by a service provider simulator. In this part, it is assumed that the preconditions of properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* will eventually hold.

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest ensures
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest)
      >
>                                          "AP1-05 (CC_Activation)"
```
"A service request remains in the inputQ until it is assigned to a service queue."

```
<∀ aServiceRequest ::
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
      >
      ensures
            < ∃ aServiceProviderSimulator where
            spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator.serviceRequest = aServiceRequest
            >
>                                          "AP1-06 (CC_Activation)"
```
"A service request remains in a service queue until it is allocated to an element of the service provider container."

In properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* `aServiceQueue` is quantified over all the service queues associated with `ServiceCategory` instances in the `scContainer`. This quantification is not shown in order to make the property specifications less cluttered.

In Chapter 4, Section 4.3.4.4, it was stated that the SLOOP **leads-to** relation can be derived by applying the same inference rules as specified for the UNITY **leads-to** relation. Those inference rules were presented in Chapter 2, Section 2.5.5. The correctness arguments for property *AL3-01(CC_SimulationActivation)* uses the first two inference rules, viz.

❑
$$\frac{p \text{ ensures } q}{p \rightarrow q}$$

❑
$$\frac{p \rightarrow q,\ q \rightarrow r}{p \rightarrow r} \quad \text{(transitivity)}$$

The **ensures** relation in the properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* can therefore be replaced with **leads-to** relations. Applying the transitivity rule on the **leads-to** relations in these properties allows one to deduce the following:

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest leads-to
      < ∃ aServiceProviderSimulator where
      spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator serviceRequest = aServiceRequest
      >
>
```

Thus, once a service request is present in the `inputQ`, it will eventually be serviced by a service provider simulator. The above example demonstrates how a **chain of eventualities** is used in the correctness arguments of property *AL3-01 (CC_SimulationActivation)*. The concept of proof by eventuality chains is described in [MaPn81b] as an approach "based on establishing a chain of eventualities that by transitivity leads to the ultimate establishing of the desired goal".

The results of properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* are reused here. Their correctness must be shown separately. As an example, the correctness arguments for property *AP1-06 (CC_Activation)* are presented in Section 7.3.3.1. This concludes part A of the correctness argument.

Part B:
It now remains to be shown that the preconditions of properties *AP1-05 (CC_Activation)* and *AP1-06 (CC_Activation)* will eventually hold.

The precondition of property *AP1-05 (CC_Activation)* specifies that the service request associated with a **new** connection should be present in the `inputQ`. It is evident from safe liveness property *AP1-01* of the `p_generateEvent:target:` method of the `CommsProviderSimulator` class, which was given at the beginning of this section, that the associated service request is entered into the `inputQ` when a new connection is established. The precondition of property *AP1-05(CC_Activation)* therefore holds once the user has connected successfully to the service centre, which means that its postcondition will eventually hold.

The precondition of property *AP1-06 (CC_Activation)* specifies that the service request has to be entered into a service queue. This follows directly from the postcondition of property *AP1-05 (CC_Activation)*. This concludes the second part of the correctness argument.

In this section it has been demonstrated how the properties of a **leads-to** relation can be utilised in the correctness arguments of a liveness property. It has illustrated the application of **eventuality chains** in informal liveness property proofs. It has also demonstrated that when reusing other correctness properties in the correctness arguments of a **liveness** property, it is important to show that the preconditions of the **properties** being reused will eventually become true. It is only if the preconditions do eventually become true that the postconditions can hold and that progress can take place. Preconditions play an equally significant role when the `postconditions:` construct is used in a correctness property. That is the topic of the Section 7.3.3.1, which covers the correctness arguments of one of the precedence properties. The next section discusses various precedence properties.

7.3.3 Precedence properties

In Chapter 5 three types of precedence properties were listed, viz. safe liveness, absence of unsolicited response and fair responsiveness. Informal correctness arguments are now given for examples of each of these correctness property types. The examples have been chosen to highlight various aspects of correctness reasoning. The safe liveness property example illustrates the usage of the `postconditions:` and `postconditions:withArguments:` constructs in correctness arguments. The absence of unsolicited response example is used to demonstrate how the distinctive characteristics of an **ensures** relation can be used in correctness arguments. Finally, the fair responsiveness example shows how correctness arguments are used to check that the classes selected for a system indeed satisfy the correctness properties as specified for the system under development.

7.3.3.1 Using a safe liveness property to highlight the impact of the postconditions: and postconditions:withArguments: constructs on correctness arguments

The purpose of this section is to describe the impact of **postconditions:** and **postconditions:withArguments:** constructs on correctness arguments. The discussion highlights the importance of showing that the **preconditions** of the methods referenced in the `postconditions:` and `postconditions:withArguments:` constructs are satisfied. Property *AP1-06 (CC_Activation)* is used in this example.

```
<∀ aServiceRequest ::
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
      >
      ensures
            < ∃ aServiceProviderSimulator where
            spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator serviceRequest = aServiceRequest
            >
>                                          "AP1-06 (CC_Activation)"
```

"A service request remains in a service queue until it is allocated to an element of the service provider container."

In property *AP1-06(CC_Activation)* `aServiceQueue` is quantified over all service queues associated with `ServiceCategory` instances in the `scContainer`. This quantification is not shown in order to make the property specifications less cluttered. Error conditions are not described at this level of refinement.

The strategy to be followed for the correctness arguments

It has to be shown that:
A) once a service request is present in a service queue, it remains in the service queue unless the service request is allocated to a service provider simulator (the safety part) and
B) eventually a service request is allocated to a service provider simulator (the liveness part).

The strategy for the informal proof of part A is as follows:
A1) Find all the correctness properties that imply the removal of a service request from a service queue. In this case only one such property is found, viz. the *DP1-01(ServiceCategory)* property.
A2) Show that the service request is assigned to a service provider simulator when it is removed from the service queue.
A3) Since the `postconditions:withArguments:` construct is used to specify the assignment of a service request to a service provider simulator, it needs to be shown that the preconditions of the corresponding method will hold when the latter starts its execution.

A4) The action in A3 is applied recursively until no further `postconditions:` or `postconditions:withArguments:` constructs are found.

The strategy for the informal proof of part B is as follows:
Once the correctness arguments of Part A have been presented, it will be evident that there is only one method which removes a service request from a service queue, viz. the `p_execute` method of the `ServiceCategory` class. It will also be clear that when the service request is removed from the service queue, it is assigned to a service provider simulator (an element of the `spAgentContainer`) in a single atomic action. It now only remains to be shown that the preconditions of the safe liveness correctness property of the `p_execute` method will eventually hold.
B1) The first predicate of the preconditions of this property requires a non-empty `serviceQ`. It therefore needs to be argued that a service request will be present in the `serviceQ`.
B2) The second predicate of the safe liveness property of the `p_execute` method specifies that a service provider simulator will eventually be willing to accept the service request. It therefore needs to be shown that:
B2.1) Each service provider simulator eventually responds with the value true when the `canAcceptNextSR:` message is sent to it, provided the service category passed as parameter matches one of the categories serviced by the service provider simulator.
B2.2) A service category only sends the `canAcceptNextSR:` message to service provider simulators that have indicated their capability to service that particular service category. These service provider simulators are elements of the `spSubset` collection of the `ServiceCategory` instance.
B2.3) The `spSubset` collection of each `ServiceCategory` instance contains at least one element.

Thus, by providing correctness as outlined above, it can be concluded that any service request present in a service queue remains in that queue until it is assigned to a service provider simulator.

The correctness arguments

Part A1:
First of all the classes comprising the call centre system are inspected with the aim of finding a correctness property that implies the removal of a service request from a service queue. The only one that is found, belongs to the `p_execute` method of the `ServiceCategory` class[20]. Its safe liveness property specifies:

```
serviceQ isEmpty not ∧ self canAssignSR ensures
      self postconditions: (#assignToSP:) withArguments:
      #((serviceQ first)) ∧
      serviceQ postconditions: (#removeFirst)
                                        "DP1-01(ServiceCategory)".
```

Part A2:
The next step is to show that a service request is assigned to a service provider simulator when it is removed from the service queue. This is evident from the postconditions of the safe liveness property of the `p_execute` method. These indicate that the `assignToSP:` and `removeFirst` methods are invoked; the former to assign a service request to a service provider simulator and the latter to remove that service request from a service queue. The fact that property *DP1-01(ServiceCategory)* is an **ensures** relation, guarantees that these actions will be executed atomically.

[20] This can be verified by inspecting the statements of the classes in Appendix B. The ServiceCategory class is specified in Appendix B, Section B.10.

Inspection of the correctness properties of the `assignToSP:` method reveals that the `assignToSP:` invokes the `processServiceRequest:` method of the `ServiceProviderSimulator` instance in order to perform the actual assignment of the service request to a service provider simulator, as shown below:

```
"Total correctness property of the assignToSP: method"
sr notNil ∧ availableServiceProvider notNil results-in
      methodReturnValue = self ∧
      sr postconditions: (#serviceProvider:[21])
            withArguments: #(availableServiceProvider) ∧
      availableServiceProvider
            postconditions:(#processServiceRequest:)
            withArguments: #(sr)     "DL1-10 (ServiceCategory)"
```

The `availableServiceProvider` *macro-variable* is defined as:

```
availableServiceProvider ≡
      spSubset detect: [:each | each canAcceptNextSR:
      serviceQCategory]        .
```

It is when the `processServiceRequest:` message is sent to `availableServiceProvider` that the service request is assigned to the service provider, as is evident from the total correctness property of the `processServiceRequest:` method of the `ServiceProviderSimulator` class:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Thus, by inspecting the correctness properties of the classes comprising the call centre, one can deduce that when a service request is removed from a service queue, then it is assigned to a service provider simulator, provided the preconditions of the `assignToSP:` and `processServiceRequest:` methods are met. The next two parts of this informal proof are devoted to showing that the preconditions are indeed satisfied.

Part A3:

Property *DP1-01(ServiceCategory)*, which describes the behaviour of the `p_execute` method, uses the `postconditions:withArgument:` construct to convey the fact that a service request is assigned to a service provider simulator. In order to ensure that the allocation will be successful, it has to be shown that the preconditions of the `assignToSP:` method will always be satisfied when the latter is invoked from within the `p_execute` method.

[21] The serviceProvider: method of the ServiceRequest instance sets the serviceProvider attribute of that instance to the value specified in the argument of the method. In this case it refers to the service provider simulator that will be processing the service request. Refer to Appendix B, Section B.9 for details of this method.

In fact, it can be said that the preconditions of the `p_execute` method of the `ServiceCategory` class were designed with the aim of ensuring that the preconditions of any other methods invoked by the statements of the `p_execute method` would be satisfied vacuously. This is possible because the `p_execute` method contains only one statement that could change the values of the predicates appearing in the precondition of property *DP1-01(ServiceCategory)* and that is the statement that invokes the method(s) that have the same preconditions as the `p_execute` method itself. At this stage it is assumed that the preconditions of the `p_execute` method hold. The correctness arguments to prove that they do indeed hold, are presented in part B. The predicates of the preconditions of the `assignToSP:` method are now considered one by one.

The first predicate of property *DL1-10(ServiceCategory)* of the `p_execute` method (given above) specifies that `sr` should not be nil, where `sr` is the message argument in the message pattern. Thus, `assignToSP:` should be invoked with a non-nil parameter. Upon inspection of property *DP1-01(ServiceCategory)* describing the behaviour of the `p_execute` method, it emerges that the argument that is passed to the `assignToSP:` method is `serviceQ first` (this is clear from the `postconditions:withArguments:` constructs used in that correctness property). The first element of `serviceQ` is guaranteed not to be nil by the `serviceQ isEmpty not` precondition of the *DP1-01(ServiceCategory)* correctness property of the `p_execute` method.

The second precondition of the total correctness property of the `assignToSP:` method specifies that `availableServiceProvider notNil` has to hold. This means that there has to be at least one service provider in `spSubset` that can accept a new service request, as is evident from the definition of the `availableServiceProvider` *macro-variable* listed above.

The second precondition of property *DP1-01(ServiceCategory)* of the `p_execute` method contains the following predicate: `self canAssignSR`. The total correctness property of the `canAssignSR` method of the `ServiceCategory` class ensures that a value of true is only returned if there is at least one service provider simulator in `spSubset` that can accept a new service request, as can be seen below:

```
"Total correctness property of the canAssignSR method"
true results-in methodReturnValue =
      (spSubset detect:
      [:each | each canAcceptNextSR: serviceQCategory]
      ifNone: [nil] ) notNil          "DL1-05(ServiceCategory)"
```

Thus, if the preconditions of the property *DP1-01(ServiceCategory)* are satisfied, then it implies that the `canAssignSR` method returns true. As is evident from the above, this means that the preconditions of the `assignToSP:` method are also satisfied.

Part A4:
As stated earlier, it is not sufficient to check that the preconditions of the methods invoked by the `p_execute` method are satisfied. One also has to ensure that the preconditions of methods invoked by methods invoked by the `p_execute` method are satisifed. Thus, step A3 has to be executed recursively. The `assignToSP:` method, which is called from within the `p_execute` method, in turn also invokes other methods (this is evident from the presence of the `postconditions: withArguments:` constructs in its total correctness property specification given above). One therefore needs to check that the preconditions of those methods are also satisfied in order to ensure that their postconditions will hold.

The precondition of the `serviceProvider:` method of the `ServiceRequest` class[22] is specified as `true` and is therefore satisfied vacuously.

The total correctness property of the `processServiceRequest:` method is as follows:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

The first predicate involves the `categoryIndex` instance variable of the `ServiceProviderSimulator` class. The latter does not export any methods to modify this variable, so it is the responsibility of the `ServiceProviderSimulator` instance to ensure that the value of `categoryIndex` is restricted to the specified range. This is guaranteed by the class invariant shown below:

```
invariant categoryIndex ≥ 0 ∧
      categoryIndex < nrOfCategoriesServed
                          "DS2-01 (ServiceProviderSimulator)"
```

"The categoryIndex is always greater than or equal to zero and less than nrOfCategoriesServed."

The second predicate in the preconditions of the `processServiceRequest:` method specifies that `aServiceRequest` (the argument of the method)[23] should not be nil. Since the `assignToSP:` method invokes the `processServiceRequest:` method passing its *pseudo-variable* `sr` as the argument, and since the preconditions of the `assignToSP:` method in turn requires `sr` to be not nil, the value of `aServiceRequest` is guaranteed not to be nil. (The value of `sr` cannot be changed by the `assignToSP:` method itself, since the value of a *pseudo-variable* may not be changed.)

The third precondition of the total correctness property of the `processServiceRequest:` method requires that a `ServiceProviderSimulator` instance should be willing to accept `aServiceRequest`, i.e. the `canAcceptNextSR:` method has to return the value true. As shown earlier in this section, the `assignToSP:` method is only executed if its preconditions are satisfied. This means the `availableServiceProvider`[24] `notNil` predicate of the preconditions of the `assignToSP:` method will always be true when the `processServiceRequest:` method is invoked, since the latter is invoked from within one of the statements of the `assignToSP:` method and there are no statements[25] in `assignToSP:` that could change the outcome of the `canAcceptNextSR:` method before `processServiceRequest:` is invoked. It therefore follows that the preconditions of the `processServiceRequest:` method will hold when it is invoked from within the `assignToSP:` method.

---

[22] The SLOOP specification of the ServiceRequest class is given in Appendix B, Section B.9.
[23] The processServiceRequest: method is specified in Appendix B, Section B.13, where the usage of its argument is shown.
[24] The availableServiceProvider macro definition was discussed in part A2.
[25] This can be verified by checking the statements of the assignToSP: method in Appendix B, Section B.10.

The above arguments have shown informally that the service request remains in the service queue unless it is assigned to a service provider simulator. This concludes Part A of the correctness arguments.

Part B:
For Part B of the correctness argument it needs to be shown that the postconditions of the *AP1-06(CC_Activation)* safe liveness property will eventually be satisfied, provided the precondition holds. Thus, it has to be shown that there exists a statement which will assign an element of a service queue to a service provider simulator. This is guaranteed by the safe liveness property of the `p_execute` method of the ServiceCategory class. To recapitulate, this property states that:

```
serviceQ isEmpty not ∧ self canAssignSR ensures
        self postconditions: (#assignToSP:) withArguments:
        #((serviceQ first)) ∧
        serviceQ postconditions: (#removeFirst)
                                                "DP1-01(ServiceCategory)".
```

In order to ensure that the postconditions will eventually become true, the preconditions have to be satisfied eventually. This is discussed in parts B1 and B2 of the informal proof.

Part B1:
The first predicate, viz. `serviceQ isEmpty not`, follows directly from the preconditions of property *AP1-06 (CC_Activation)*. Thus it follows directly from the premise of the whole discussion.

Part B2:
The second predicate requires the `canAssignSR` method to return the value `true`. Recall that the total correctness property of the `canAssignSR` method is as follows:

```
true results-in methodReturnValue =
        (spSubset detect:
        [:each | each canAcceptNextSR: serviceQCategory]
        ifNone: [nil]) notNil          "DL1-05(ServiceCategory)"
```

Thus, the method returns true if there is at least one service provider simulator which returns `true` when the `canAcceptNextSR:` message is sent to it.

Part B2.1:
The first step is to show that each service provider simulator will eventually respond with the value `true` when it receives the `canAcceptNextSR:` message, provided that the service category that is passed as parameter is an element of the collection of service categories that it services. This is guaranteed by the liveness property specified for the `ServiceProviderSimulator` class, viz.

```
<∀ categoryIndex where 0 ≤ categoryIndex ∧
categoryIndex < nrOfCategoriesServed ::
¬(self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1)))
leads-to
        self canAcceptNextSR:
        (categoriesServed at: (categoryIndex + 1))
>                          "DL2-01 (ServiceProviderSimulator)"
```

"For any service category serviced by the service provider simulator, the service provider simulator will eventually be able to service a request from that service category."

One now needs to show that each service category only interrogates the service provider simulators that service that particular category. That is the topic of Part B2.2.

Part B2.2:
From the total correctness property of the `canAssignSR` method given at the start of Part B2, it is clear that the `ServiceCategory` instance only invokes the `canAcceptNextSR:` method on members of its `spSubset` collection. This is the collection that contains the service provider simulators that will service requests from the service queue belonging to the `ServiceCategory` instance. This is evident from the following total correctness properties of the `ServiceProviderSimulator` and `ServiceCategory` classes respectively.

Property *DL1-05* of the `ServiceProviderSimulator` class specifies the behaviour of the simulator when it registers itself with the `ServiceCategory` instances that it services. This property specifies, *inter alia*, that the `ServiceProviderSimulator` instance invokes the `addSP:` method of the `ServiceCategory` class for every `ServiceCategory` instance that it services:

```
"Total correctness property of the
registerServiceProvider:using: method"
true results-in
      methodReturnValue = self ∧
      <∀aServiceCategory where
      scContainer includes: aServiceCategory ∧
      aServiceCategory servicedBy: serviceProviderCategory ::
      aServiceCategory postconditions: (#addSP:)
      withArguments: #(self) ∧
      categoriesServed includes:
            (aServiceCategory serviceCategory)
      > ∧
      nrOfCategoriesServed = categoriesServed size ∧
      categoryIndex ≥ 0  "DL1-05 (ServiceProviderSimulator)"
```

In turn, property *DL1-09* of the `ServiceCategory` class specifies that the `addSP:` method adds the service provider simulator passed as parameter (via the `anSP` *pseudo-variable*) to the `spSubset` collection:

```
"Total correctness property of the addSP: method"
anSP notNil results-in
      methodReturnValue = self ∧
      spSubset includes: anSP        "DL1-09 (ServiceCategory)"
```

Thus, the `spSubset` of each `ServiceCategory` instance contains all the service provider simulators that are able to process service requests belonging to that particular service category.

Part B2.3:
It now remains to be shown that the `spSubset` collection of each `ServiceCategory` instance will have at least one element. This follows directly from property *AS2-10(CC-Activation)*, which specifies that the service provider subset of each service category contains at least one element.

AS2-10. The service provider subset of each service category contains at least one (simulated) service provider instance.

This concludes the correctness arguments for part B and thus for property *AP1-06 (CC_Activation)*. In this section the emphasis has been on the `postconditions:` and `postconditions:withArguments:` constructs. It was demonstrated how these constructs **highlight** the fact that **other methods are being invoked** from within the method under discussion. It was also shown what role these constructs play in correctness arguments.

Another aspect worth noting here is the role that correctness arguments play in the discovery of design flaws. In the call centre example the original version of the `canAcceptNextSR:` method returned true if no service request was assigned to the service provider simulator. The method did not take any service categories into account. However, it was while working through the correctness arguments of property *AP1-06* that it became clear that in the original version of the design, starvation of a specific service category was possible.

That could have happened if there were multiple `ServiceCategory` instances and whenever a particular `ServiceCategory` instance executed its `p_execute` method, then the `ServiceProviderSimulators` would be busy with a service request from one of the other categories. Thus, the design flaw was discovered while trying to prove that the `canAcceptNextSR:` method will eventually return true when invoked by a specific `ServiceCategory` instance (i.e. while trying to prove part B2 of the above correctness arguments). As a result the design was modified to ensure that the service categories were serviced in a round robin fashion, unless the associated service queues were empty.

7.3.3.2 Using an absence of unsolicited reponse property to demonstrate how the characteristics of an ensures relation can be used in correctness arguments

In this section the focus is on the significance of the **ensures** relation in correctness arguments. This relation is distinguished from similar relations such as **leads-to** and **until** by the fact that the transition from the state where the preconditions are holding to where the postconditions are holding occurs in a single atomic step. If a correctness property contains an **ensures** relation, one is therefore guaranteed that the precondition will hold up until the point when the postconditions start to hold. This concept is illustrated via the correctness arguments of property *AP2-01 (CC_Activation)*. This property is as follows:

```
<∀ spAgentContainerElement where
spAgentContainer includes: spAgentContainerElement ::
    <∀ aServiceRequest ::
      < ∃ aServiceQueue ::
      aServiceQueue includes: aServiceRequest precedes
      spAgentContainerElement serviceRequest = aServiceRequest
      >
    >
>                                                    "AP2-01 (CC_Activation)"
```

"A service request is assigned to an element of the service provider container only if the service request has been enqueued in a service queue and has remained in the queue until it was assigned to the service provider container element."

The strategy to be followed for the correctness arguments

It has to be shown that :
A) a service request is only allocated to a service provider simulator if the former has been enqueued in a service queue and
B) once a service request is enqueued in a service queue, it remains there until it is allocated to a service provider simulator.

The correctness arguments

Part A:
A service request is allocated to a service provider simulator via the `processServiceRequest:` method of the `ServiceProviderSimulator` class, as is evident from the total correctness property of this method:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
results-in
      methodReturnValue = self ∧
      serviceRequest = aServiceRequest ∧
      newEventRequired ∧
      categoryIndex = (x + 1) \\ nrOfCategoriesServed
>                          "DL1-06 (ServiceProviderSimulator)"
```

Upon inspection of the correctness properties of the classes used in the call centre system, it is found that the `processServiceRequest:` method is invoked only by the `assignToSP:` method of the `ServiceCategory` class. In the total correctness property of the `assignToSP:` method the *pseudo-variable* `sr` refers to the service request that is received as argument of the `assignToSP:` method. In turn, the `assignToSP:` method passes the value of `sr` as argument to the `processServiceRequest:` method.

```
"Total correctness property of the assignToSP: method"
sr notNil ∧ availableServiceProvider notNil results-in
      methodReturnValue = self ∧
      sr postconditions: (#serviceProvider:[26])
            withArguments: #(availableServiceProvider) ∧
      availableServiceProvider
            postconditions:(#processServiceRequest:)
            withArguments: #(sr)     "DL1-10 (ServiceCategory)"
```

The `availableServiceProvider` *macro-variable* is defined as:

```
availableServiceProvider ≡
      spSubset detect: [:each | each canAcceptNextSR:
      serviceQCategory]        .
```

The value of `sr` is determined by the `p_execute` method which invokes the `assignToSP:` method. From the precedence property of the `p_execute` method of the `ServiceCategory` class it is clear that the service request that is assigned to a service provider simulator is taken from a service queue:

```
"Safe liveness property of the p_execute method:"
serviceQ isEmpty not ∧ self canAssignSR ensures
      self postconditions: (#assignToSP:) withArguments:
      #((serviceQ first)) ∧
      serviceQ postconditions: (#removeFirst)
                                    "DP1-01(ServiceCategory)".
```

The fact that property *DP1-01(ServiceCategory)* contains an **ensures** relation, guarantees that the service request is removed from the `serviceQ` and assigned to the service provider simulator in

[26] The serviceProvider: method of the ServiceRequest instance sets the serviceProvider attribute of that instance to the value specified in its argument. In this case it refers to the service provider simulator that will be processing the service request. Refer to Appendix B, Section B.9 for details of this method.

a single atomic step. Thus, a service request that is assigned to a service provider simulator, is always taken from a service queue. This concludes the first part of the correctness argument.

Part B:
The second part of the correctness argument reuses the results of property *AP1-06 (CC_Activation)*. In the section 7.3.3.1 it was shown that:

```
<∀ aServiceRequest ::
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest
      >
      ensures
            < ∃ aServiceProviderSimulator where
            spAgentContainer includes: aServiceProviderSimulator ::
            aServiceProviderSimulator serviceRequest = aServiceRequest
            >
>                                                    "AP1-06 (CC_Activation)"
```

"A service request remains in a service queue until it is allocated to an element of the service provider container."

Again the property being reused contains an **ensures** relation, which guarantees that the service request will remain in the service queue until it is assigned to a service provider simulator. Property *AP2-01 (CC_Activation)* follows from parts A and B of the correctness argument.

The example in this section has highlighted how the characteristics of a relation such as **ensures** are used in correctness arguments. When an **ensures** relation appears in a correctness property, the software designer can safely assume that once the preconditions hold, there can be no **interference** which could affect the postconditions specified for the property.

7.3.3.3 Using a fair responsiveness property to illustrate the importance of showing via correctness arguments that the selection of the constituent classes of a system will indeed result in the behaviour as described in the specification of the system

This section illustrates a specific aspect of the role of correctness arguments in the SLOOP method, viz. that the correctness arguments are used to check **informally** that the **actual behaviour** of the system as implied by the **constituent classes** of the SLOOP program matches the **specified behaviour** of the system as implied by the correctness properties of the system. The correctness arguments given below for property *AP3-01 (CC_Activation)* exemplify this aspect of correctness reasoning.

When a service request reaches the head of the `inputQ`, the `ServiceCategoryAllocator`[27] class is used to categorise the service request. This procedure can be quite elaborate, e.g. in the case where a database has to be consulted in order to obtain specific information. It is therefore possible that the `ServiceCategoryAllocator` class could be designed to use parallel statements to perform the categorisation of the service request. Thus, it might not necessarily be an atomic action. One design option is to start the categorisation in a FIFO order, but to allow the service requests to be added to the appropriate service queues as the categorisation for each service request finishes (which might not necessarily correspond to the start order).

Since the parallel statements may be executed in any order, it is possible that even if the service requests are categorised in the order in which they were entered into the `inputQ`, they may still not be assigned to the service queues in that order. However, the requirements analysis states that the service requests should be assigned to the service queues in the order in which the connections were established, which is reflected by property *AP3-01 (CC_Activation)*. When

[27] The ServiceCategoryAllocator class is defined in Appendix B, Section B.8.

selecting the class which has to perform the categorisation of the service requests, it has to be shown that the properties of this class do not violate the properties of the system under development. Property *AP3-01 (CC_Activation)*, which represents one of the correctness properties of the call centre system, is used in the example below to illustrate how the selection of the `ServiceCategoryAllocator` class preserves property *AP3-01(CC_Activation)*.

First of all property *AP3-01(CC_Activation)* is specified more formally. Its specification has two parts: the first part specifies that **if** a service request is added to the `inputQ`, then it is always added to the **end** of the `inputQ` and the second part specifies that **if** `aServiceRequestX` is ahead of `aServiceRequestY` in the `inputQ`, then `aServiceRequestX` **will always** be removed first from the `inputQ`.

```
<∀ aServiceRequestX where
      ¬ (inputQ includes: aServiceRequestX) ::
      ¬ (inputQ includes: aServiceRequestX) unless
      inputQ last = aServiceRequest
 > ∧
<∀ (aServiceRequestX, aServiceRequestY) where
      aServiceRequestX ~~ aServiceRequestY ∧
      inputQ includes: aServiceRequestX ∧
      inputQ includes: aServiceRequestY ::

      inputQ indexOf: aServiceRequestX <
      inputQ indexOf: aServiceRequestY ensures

      ¬ (inputQ includes: aServiceRequestX) ∧
      inputQ includes: aServiceRequestY ∧
      <∃ aServiceQueue :: aServiceQueue includes: aServiceRequestX
      >
>                                              "AP3-01 (CC_Activation)"
```

"Service requests are added and removed from the input queue on a First In First Out basis."

The significance of the **unless** and **ensures** relations in the above property is as follows: The **unless** relation indicates that **if** the service request is added to the `inputQ`, then it will be added to the end of the queue, but it does not guarantee that a service request **will** eventually be added to the `inputQ`. The **unless** relation therefore specifies the behaviour of the system **if** a service request has to be added to the `inputQ` (it describes a **safety** aspect of the system behaviour).

In contrast, the **ensures** relation has both safety and liveness characteristics. The **safety** aspect specifies that **if** a service request is ahead of another service request in the `inputQ` then it will always be processed first. The **liveness** aspect specifies that if a service request is present in the `inputQ`, then it **will eventually** be removed from the `inputQ` and added to a service queue.

Error conditions are not specified at this level of refinement. Service queues are quantified over all service queues associated with service categories in the `scContainer`. This quantification is not shown in order to make the specification less cluttered.

The informal proof of this property is now presented, based on the assumption that the `ServiceCategoryAllocator` is selected as the class to perform the categorisation of the service requests and the allocation of service requests to service queues. By showing the correctness of the above property based on this assumption, it can be deduced that the `ServiceCategoryAllocator` class does not violate the above property.

The strategy to be followed for the correctness arguments

It has to be shown that
A) **if** a service request is added to the `inputQ`, then it is always added to the **end** of the `inputQ`,
B) if `aServiceRequestX` and `aServiceRequestY` are both present in the `inputQ` and `aServiceRequestX` is ahead of `aServiceRequestY`, then `aServiceRequestX` is removed from the `inputQ` and allocated to a service queue **before** `aServiceRequestY` and
C) **eventually** `aServiceRequestX` is allocated to a service queue while `aServiceRequestY` remains in the `inputQ`.

In turn, part B also has three parts. It is shown that
B1) a service request is always removed from the head of the `inputQ`,
B2) the relative ordering between elements of the `inputQ` is maintained and
B3) when a service request is removed from the `inputQ`, it is added to a service queue.

The correctness arguments

Part A:
Inspection of the classes that constitute the call centre system yields the `CommsProviderSimulator`[28] class as the only one containing a method which adds a service request to the `inputQ`. Safe liveness property *AP1-01(CommsProviderSimulator)* of the `p_generateEvent: target:` method specifies the following:

```
generatingEvent ∧ idleConnection notNil ∧
¬(inputQ includes:(idleConnection serviceRequest)) ensures
        ¬generatingEvent ∧ newEventRequired ∧
        inputQ last = (idleConnection serviceRequest) ∧
        idleConnection postconditions: (#assign)
                                "AP1-01 (CommsProviderSimulator)"
```

where

```
idleConnection ≡ self getIdleConnection: userConnections
```

Thus, when a service request is added to the `inputQ`, it is always added at the end. This concludes part A of the informal proof.

Part B:
The next step is to show that if `aServiceRequestX` and `aServiceRequestY` are both present in the `inputQ` and `aServiceRequestX` is ahead of `aServiceRequestY`, then `aServiceRequestX` is removed from the `inputQ` and allocated to a service queue **before** `aServiceRequestY`.

Part B1:
It must first be shown that a service request is always removed from the head of the `inputQ`. Another inspection of the call centre classes reveals that the only method that results in the removal of a service request from the `inputQ`, is the `p_allocate:from:` method of the `ServiceCategoryAllocator`[29] class. The relevant safe liveness property is shown below.

[28] The CommsProviderSimulator class is specified in Appendix B, Section B.6.
[29] The ServiceCategoryAllocator class is defined in Appendix B, Section B.8.

```
<∀ aServiceRequest where
¬(inputQ isEmpty) ∧ inputQ first == aServiceRequest ::
      aServiceRequest serviceRequestCategory notNil ∧
      aServiceRequest serviceQ isNil ∧
      < ∃ aServiceCategory where
      scContainer includes: aServiceCategory ::
            aServiceCategory serviceQCategory =
            aServiceRequest serviceRequestCategory
      >
 ensures
      self postconditions: (#assignToSQ:using:)
      withArguments: #(aServiceRequest scContainer) ∧
      ¬categorising ∧
      ¬(inputQ includes: aServiceRequest)
>                                    "DP1-04(ServiceCategoryAllocator)"
```

From the above property it is clear that the service request being dealt with here is the one at the head of the `inputQ`. (The variable `aServiceRequest` is defined as being equivalent to `inputQ first`.) The postconditions of property *DP1-04(ServiceCategoryAllocator)* indicate that `aServiceRequest` is no longer an element of `inputQ` after the `p_allocate:from:` method has completed its execution. Since this is the only method that results in the removal of a service request from the `inputQ`, it means that service requests are always removed from the head of the `inputQ`, which concludes part B1 of the correctness arguments.

Part B2:
The `inputQ` is created as an instance of the Smalltalk `OrderedCollection` library class. One of the properties of that class is that it maintains the relative ordering of its elements. There are also no statements in the `CallCentreSimulation` program that add or remove elements from the `inputQ` other than those described in parts A and B1. This means that if `aServiceRequestX` is ahead of `aServiceRequestY` in the `inputQ`, then `aServiceRequestX` will always be removed from the `inputQ` before `aServiceRequestY`.

PartB3:
It now remains to be shown that when a service request is removed from the `inputQ`, then it is added to a service queue. In property *DP1-04(ServiceCategoryAllocator)*, which was presented in part B1, it is stated that when `aServiceRequest` is removed from the `inputQ`, then the `assignToSQ:using:` method of the `ServiceCategoryAllocator` class is also executed by the same statement (property *DP1-04(ServiceCategoryAllocator)* is an **ensures** relation).

The `assignToSQ:using:` method is invoked using `aServiceRequest` as one of its arguments. One therefore has to check whether the execution of this method results in the allocation of `aServiceRequest` to a service queue. This is indeed the case, as is evident from the total correctness property of the `assignToSQ:using:` method of the `ServiceCategoryAllocator` class (the *pseudo-variable* `serviceRequest` used in the `assignToSQ:using:` method corresponds to the *pseudo-variable* `aServiceRequest` passed as an argument to the `assignToSQ:using:` method):

```
      "Total correctness property of the assignToSQ:using: method"
      serviceRequest serviceQ isNil ∧
      serviceRequest serviceRequestCategory notNil ∧
      match notNil results-in
            methodReturnValue = self ∧
            serviceRequest serviceQ notNil ∧
            serviceRequest serviceRequestCategory notNil ∧
            (match serviceQ) includes: serviceRequest
                                    "DL1-04(ServiceCategoryAllocator)"
```

The *macro-variable* `match` is defined as:

```
match ≡ scContainer detect: [:each | each serviceQCategory =
      serviceRequest serviceRequestCategory] ifNone: [nil]
```

The preconditions of the `assignToSQ:using:` method are also preconditions of the `p_allocate:from:` method. Since there is only one statement in the `p_allocate:from:` method and that is the one invoking the `assignToSQ:using:` method, these preconditions will therefore still hold when the `assignToSQ:to:` method is invoked from within the `p_allocate:from:` method. The statements of the `p_allocate:from:` method are shown below for easy reference.

```
"The statements of the p_allocate:from: method"
parallel
self assignToSQ: serviceRequest using: scContainer \+
categorising := false \+
inputQ removeFirst
      if serviceRequest notNil and:
      [serviceRequest serviceRequestCategory notNil and:
      [serviceRequest serviceQ isNil]]
end-parallel
```

Thus, if the preconditions of the `p_allocate:from:` method hold when the latter is executed, then the service request at the head of the `inputQ` will be removed and the `assignToSQ:using:` method will be executed. As shown above, the preconditions of the `assignToSQ:using:` method will hold when this method is invoked, therefore the postconditions of the `assignToSQ:using:` method are guaranteed to hold after its execution (i.e. the service request will have been added to a service queue).

Thus, since

- a service request is always removed from the head of the `inputQ`,
- the relative ordering of the elements of the `inputQ` is always maintained and
- the removal of a service request from the `inputQ` coincides with its allocation to a service queue,

the following is implied:

If the index of `aServiceRequestX` in the `inputQ` is less than the index of `aServiceRequestY` in the `inputQ`, then `aServiceRequestX` is allocated to a service queue before `aServiceRequestY`. This concludes part B of the correctness arguments.

Part C:

Part C of the correctness argument reuses the results of property *AP1-05 (CC_Activation)*.

```
<∀ aServiceRequest where inputQ includes: aServiceRequest ::
      inputQ includes: aServiceRequest ensures
      ¬(inputQ includes: aServiceRequest) ∧
      < ∃ aServiceQueue :: aServiceQueue includes: aServiceRequest)
      >
>                                                "AP1-05 (CC_Activation)"
```

"A service request remains in the inputQ until it is assigned to a service queue."

The above property implies that each service request **will eventually** be removed from the `inputQ`. If the index of `aServiceRequestX` is less than that of the index of `aServiceRequestY`, then `aServiceRequestX` will eventually be removed from the `inputQ`, while `aServiceRequestY` will still be an element of `inputQ`. This is because the relative ordering of the elements of the `inputQ` always remains the same (as was shown in part

B2) and service requests are always removed from the head of the `inputQ` (as was demonstrated in Part B1). This concludes the correctness argument for part C and thus also for property *AP3-01 (CC_Activation)*.

By presenting the above correctness arguments based on the assumption that the `ServiceCategoryAllocator` class is used to perform the categorisation of the service requests and the allocation of these requests to service queues, it is implied that this class does not violate this property. The correctness properties specified for the system under development are therefore used not only at the **beginning** of the design phase in order to **aid the selection** of the right constituent classes of the system, but also at the **end** of the design phase to **confirm** the correctness of their selection.

7.4 Deriving SLOOP statements from correctness properties

It is now shown how correctness properties can be used to derive SLOOP statements. The example that was introduced very briefly in Chapter 4, Section 4.2.3, is used to demonstrate the concepts.

The functionality of the required system is as follows: A dispatcher system must be developed which acts as a generic transformer of objects, receiving them from a producer, performing some transformation on the received objects and then dispatching them to a consumer. The dispatcher has to queue the objects in a FIFO order until the consumer is ready to receive the next object. The dispatcher keeps a record of the maximum length ever reached by the queue managed by the dispatcher. The dispatcher ensures that the size of this queue never exceeds a specified maximum value.

The following Dispatcher class attributes can be identified:

`bufferedElements`	This attribute refers to the FIFO queue managed by the Dispatcher class.
`consumer`	This attribute refers to the instance of the Consumer class to which the Dispatcher instance will dispatch the elements in its `bufferedElements` queue.
`maximumRecordedLength`	This attribute represents the maximum length ever reached by the `bufferedElements` queue.
`maximumAllowedLength`	This attribute represents the maximum length that the `bufferedElements` queue may ever reach.
`newElement`	This attribute refers to an object which has been received from the Producer instance, but which has not yet been added to the `bufferedElements` queue.

The correctness properties below specify the required behaviour of the Dispatcher class:

invariant `newElement notNil ⇒ ¬ self readyToReceiveElement`
"AS2-01 (Dispatcher)"

"The dispatcher will never indicate that it is ready to accept a new element from the producer if it has not yet added the element passed to it on a previous occasion to the bufferedElements queue."

invariant `bufferedElements size <= maximumAllowedLength`
"AS2-02 (Dispatcher)"

"The current length of the bufferedElements queue is always less than or equal to the maximumAllowedLength."

```
invariant maximumRecordedLength >= bufferedElements size
                                                    "AS3-01 (Dispatcher)"
```

"The maximumRecordedLength of the bufferedElements queue is always greater than or equal to the current queue size."

```
< ∀ k where 0 <= k ∧ k <= maximumAllowedLength ::
maximumRecordedLength = k unless maximumRecordedLength > k
>                                                   "AS4-01 (Dispatcher)"
```

"The maximumRecordedLength of the bufferedElements queue is non-decreasing."

```
< ∀ anElement where newElement = anElement ::
newElement = anElement unless
      self postconditions: (#transform:) withArguments: #(anElement) ∧
      bufferedElements includes: anElement ∧ newElement isNil
>                                                   "AS4-02 (Dispatcher)"
```

"Once the dispatcher has accepted a new object from the producer, it remains a new object unless it is transformed and added to the bufferedElements queue."

```
< ∀ anElement where bufferedElements includes: anElement ::
bufferedElements includes: anElement unless
      consumer postconditions: (#pass:)  withArguments: #(anElement)
>                                                   "AS4-03 (Dispatcher)"
```

"Once an object is added to the bufferedElements queue, it remains there unless it is passed to the consumer."

```
<∀ anElementY) where
      ¬ bufferedElements includes: anElementY ::
      ¬ bufferedElements includes: anElementY unless
      bufferedElements last = anElementY
 >                                                  "AS4-04 (Dispatcher)"
```

"If an object is added to the bufferedElements queue, it is always added to the end of the queue."

```
<∀ (anElementX, anElementY) where
      anElementX ~~ anElementY ∧
      bufferedElements includes: anElementX ∧
      bufferedElements includes: anElementY ::

      bufferedElements indexOf: anElementX <
      bufferedElements indexOf: anElementY unless

      ¬ bufferedElements includes: anElementX) ∧
      bufferedElements includes: anElementY
>                                                   "AS4-05 (Dispatcher)"
```

"Objects are removed from the bufferedElements queue in the order that they are added to the queue."

The precedence properties are as follows:

```
< ∀ anElement where newElement = anElement ::
newElement = anElement ∧
bufferedElements size < maximumAllowedLength ensures
      self postconditions: (#transform:) withArguments: #(anElement) ∧
      bufferedElements includes: anElement ∧ newElement isNil
>                                                   "AP1-01 (Dispatcher)"
```

"If the dispatcher has received a new object from the producer and the size of the bufferedElements queue is less than its maximumAllowedLength, these conditions continue to hold until the new object is transformed and added to the bufferedElements queue."

```
< ∀ anElement where bufferedElements includes: anElement ::
bufferedElements includes: anElement ∧
consumer readyToReceiveElement ensures
        consumer postconditions: (#pass:) withArguments: #(anElement) ∧
        ¬ bufferedElements includes: anElement
>                                                    "AP1-02 (Dispatcher)"
```

"Once the consumer is ready to receive an object and the bufferedElements queue is non-empty, these conditions continue to hold until an element of the bufferedElements queue is removed from the queue and passed to the consumer."

The methods of the various classes of a system are defined once the design phase correctness properties have been specified. The behaviour described by the correctness properties yields the necessary information to derive the methods of the classes. The liveness and/or precedence properties provide the basis for deriving the parallel methods. The infinitely often execution of the statements of the parallel methods of the various classes has to result in the desired progress being made.

In the Dispatcher class example, properties *AP1-01* and *AP1-02* specify the progress that needs to be made by the Dispatcher class instance. Briefly, if the latter has accepted a new element from the Producer instance, then the new element should be transformed and added to the `bufferedElements` queue once there is space in that queue. In addition, the elements in the `bufferedElements` queue should be passed to the Consumer instance whenever the latter is ready to accept them. The above describes the crux of the functionality of the Dispatcher class. One or more parallel methods can be defined to contain the parallel statements that realise this functionality. In the case of the Dispatcher class a single parallel method called `p_dispatch` suffices, because the functionality of this class is very simple.

A useful heuristic for deriving parallel statements from properties of the form "*p* **ensures** *q*", is to populate the *if* clause using the information in the conjuncts in *p* and to use the conjuncts in *q* to determine the state changes. Note however that the conjuncts in the correctness property are not mapped to expressions in the SLOOP statement in a mechanical way. For example, in property *AP1-01* universal quantification is used and there are therefore references to `anElement`. The introduction of the variable `anElement` is necessary in order to be able to refer to the old value of the variable `newElement` (i.e. prior to the execution of the statement) in predicate *q*. Due to the evaluation order of a SLOOP parallel statement (discussed in Chapter 4, Section 4.3.6.3), it is not necessary to have such a variable in the SLOOP statement.

Thus, the aim is to devise a statement which, if executed infinitely often, will satisfy correctness property *AP1-01*. We therefore have the following statement in the `p_dispatch` method:

```
self transform: newElement \+
bufferedElements add: newElement \+
newElement := nil
    if newElement notNil and:
    [bufferedElements size < maximumAllowedLength]    "Statement S1"
```

From property *AP1-02* we have the following:

```
consumer pass: anElement \+
bufferedElements remove: anElement
```

```
    if consumer readyToReceiveElement and:
    [bufferedElements includes: anElement]          "Statement S2"
```

At this stage statement *S2* still refers to `anElement`. Property *AP1-02* does not provide any additional information regarding the identity of `anElement`. It will be necessary to consider the other correctness properties before statement *S2* can be refined further.

The derivation of the above two statements demonstrates the advantages of the SLOOP computational model. These statements are designed without having to be concerned about location counters. For example, when specifying statement *S2*, there is no need to consider the location counter of the consumer object. The only important issue is to identify the conditions that need to be satisfied in order for statement *S2* to produce the desired result. Since each parallel statement is executed infinitely often, the correct results will be achieved, provided its preconditions are true at some point and remain true until the statement has executed.

The above statements take care of the liveness and precedence properties. However, the safety properties have to be considered as well. They provide the necessary information for the refinements of these statements.

Property *AS4-04* prescribes how the new object should be added to `bufferedElements`, i.e. always at the end. Consequently the message to `bufferedElements` in statement *S1* is modified from

```
bufferedElements add: newElement
```

to

```
bufferedElements addLast: newElement
```

The resulting statement is as follows:

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

Property *AS3-01* (**invariant** `maximumRecordedLength >= bufferedElements size`) describes the invariant relationship between the size of `bufferedElements` and the value of `maximumRecordedLength`. When inspecting statement *S1* to determine whether its execution could ever violate this invariant, it is found that such a possibility exists with the execution of the statement component `bufferedElements addLast: newElement`. If the latter is executed, the size of `bufferedElements` is implicitly incremented by one. One therefore has to ensure that the value of `maximumRecordedLength` is updated whenever both of the following conditions are true: (1) the size of `bufferedElements` is incremented and (2) the new size will be greater than the current value of `maximumRecordedLength`.

This can be achieved by adding an additional component to statement *S1*. By making it part of the same statement, one ensures that the components will be executed as one atomic action. This yields a new statement *S1*:

```
   self transform: newElement \+
   bufferedElements addLast: newElement \+
   newElement := nil
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength]


|| maximumRecordedLength := bufferedElements size + 1
      if newElement notNil and:
      [bufferedElements size < maximumAllowedLength and:
```

```
    [bufferedElements size +1 > maximumRecordedLength] ]
                                                    "Statement S1"
```

Recall that all *if* clauses of all components of a particular statement are evaluated before any of the component parts are executed. All evaluations of `bufferedElements size` in statement *S1* will therefore yield the same result. Thereafter all the message expressions as listed in step 2 in Chapter 4, Section 4.3.6.3, are evaluated, followed by the evaluation of all message expressions and assignments as described in step 3 in Section 4.3.6.3.

As part of step 2 the following values are obtained (in any arbitrary order):
`self, newElement, bufferedElements, nil, (bufferedElements size + 1).`

As part of step 3 the following assignments and message expressions are executed (in any arbitrary order):

```
self transform: newElement
bufferedElements addLast: newElement
newElement := nil
maximumRecordedLength := bufferedElements size + 1
```

Since `newElement` was evaluated in step 2, the assignment of the value nil to the variable `newElement` in the third *component-part* does not affect the first or second *component-parts*. Similarly, because the value of `bufferedElements size + 1` was determined in step 2, the assignment to `maximumRecordedLength` is not affected by the execution of the `bufferedElements addLast: newElement` *component-part*.

Property *AS4-05* specifies that elements should only be removed from the head of `bufferedElements`, which provides us with the necessary information to replace `anElement` with a more specific description in statement *S2*. All references to `anElement` are therefore changed to `bufferedElements first`.

```
consumer pass: (bufferedElements first) \+
bufferedElements removeFirst
   if consumer readyToReceiveElement and:
   [bufferedElements size > 0]                  "Statement S2"
```

The correctness properties also yield a number of sequential methods. Property *AS2-01* results in the definition of sequential method `readyToReceiveElement`. Since each sequential method has to terminate, a total correctness property is defined for each sequential method. The `readyToReceiveElement` method has the following total correctness property resulting from property *AS2-01*:

```
true results-in methodReturnValue = (newElement isNil)
                                                "DL1-01 (Dispatcher)"
```

The need for the `transform:` sequential method is derived from properties *AS4-02* and *AP1-01*. These correctness properties do not specify the behaviour of the `transform:` method. In the Dispatcher class this method is defined to merely return the value of the receiver. In subclasses, various transformations may be defined.

The other sequential methods that are referenced in the correctness properties of the Dispatcher class belong to the OrderedCollection class and to the class of the consumer object. The statements contained in these methods are encapsulated within those classes and are therefore irrelevant to the discussion of the statements of the Dispatcher methods. Only the correctness properties of those methods are important when discussing the Dispatcher class.

While discussing the derivation of the methods of the Dispatcher class, no mention was made of properties *AS2-02*, *AS4-01*, *AS4-02* and *AS4-03*. These correctness properties specify constraints on the methods and statements of the Dispatcher class. Rather than indicating what statements should be included, they dictate what may not be included. For example, property *AS2-02* ensures that there will be no statement that will cause the size of the `bufferedElements` queue to exceed `maximumAllowedLength`.

Although property *AS3-01* ensures that `maximumRecordedLength` is always greater than or equal to the current size of the `bufferedElements` queue, it does not ensure that it is greater than or equal to any **previous** size of the `bufferedElements` queue. This additional constraint is provided by property *AS4-01*, which ensures that no statement will ever set the value of `maximumRecordedLength` to a value less than its current value.

Property *AS4-02* implies that the only time when `newElement` may be set to nil is when the object that it references is added to the `bufferedElements` queue. Consequently the Dispatcher class offers no method which would enable another object to set `newElement` to nil. Similarly, property *AS4-03* implies that the only time when an element is removed from the `bufferedElements` queue is when it is passed to the consumer.

The way in which correctness properties are used in the SLOOP method can be summarised as follows: At the start of the design phase, the repository of reusable artifacts is searched for classes that will match the requirements as outlined by the correctness properties identified during the requirements analysis phase. If new classes have to be designed, then once again these correctness properties provide the necessary information regarding the requirements of these classes. The SLOOP statements of the new classes are derived from the specified correctness properties. Once all the classes have been finalised, informal proofs of the correctness properties specified for the system under development confirm that the selected classes are indeed the correct ones.

7.5 Summary

This chapter has illustrated how the semantics of a class and its methods are conveyed by their correctness properties. It has also demonstrated how the various types of correctness properties can be reasoned about and how SLOOP statements can be derived from correctness properties. Correctness arguments were given for an example property of each type. These correctness arguments were based on other properties that were specified for the system or for the constituent classes, as well as on the SLOOP statements that realised those properties. Each example also highlighted at least one additional aspect of correctness reasoning in the SLOOP method. They were as follows:

- One of the **advantages of using macros** is the following: If a *macro-variable* is defined as the value returned by a method of another class, then the correctness properties regarding that value as defined by the target class may be reused by the client class. If the value had been assigned to an instance variable of the client class, then the latter would have had to specify its own correctness properties regarding the value of the instance variable.
- **Location counters are not considered** during correctness reasoning.
- It is also **not necessary** to be concerned about the **allocation of statements to processors**. At the design level, correctness reasoning is in terms of the **parallel statements**, each of which executes atomically.
- The role of the **computational model** in the correctness arguments is extremely important. There is **no need to consider all possible sequences of events.** It is only necessary to follow the **invocation paths** starting from each parallel statement that is activated via the *activation-section*.

- The **repeated execution of the parallel statements** selected for the program **eventually results in the postconditions** specified by the **liveness** properties, provided their preconditions hold at some point.
- The role of **preconditions** is to define the **responsibility of the client**. When showing that a specific property associated with a method is correct, the preconditions can be assumed to hold. It is merely necessary to show that the postconditions will be achieved, provided the preconditions hold. However, when that correctness property is being **reused** (for example when the corresponding method is being invoked by a client), then it is necessary to show that the preconditions will indeed hold at the time when the method is invoked. In the case where the property is used to ensure **progress**, it is necessary to show that the **preconditions will eventually hold**.
- The important aspect regarding reasoning about **total correctness** properties of sequential methods is the fact that **concurrency** does not need to be taken into account. **Statement interleaving** takes place at the level of **parallel** statements, which may invoke sequential methods. Each parallel statement executes **atomically**.
- Since a sequential statement executes as a single atomic unit, the postconditions of a total correctness property are always achieved during the execution of a **single** parallel statement. This means that the **ensures** logical relation (which requires its postconditions to be achieved via a single parallel statement) can include invocations of sequential methods.
- The distinctive properties of the **leads-to** logical relation, such as its transitive properties, can be utilised in correctness arguments.
- The **reuse** of correctness properties has several benefits. A great deal of **effort is saved** when the properties of classes can be assumed to hold **without having to reason about them from first principles each time**.
- Another benefit results from the usage of the `postconditions:` and `postconditions: withArguments:` constructs. The latter **highlights** the fact that other methods are being invoked by the current method. The usage of such constructs also allows one to **reason about the pre- and postconditions** of the methods being invoked **without** having to **repeat** those conditions in the present correctness property.
- While reusing correctness properties in correctness arguments, one should take care that there is **no circular reasoning** (i.e. in order to prove property A, property B is reused, but property B depends on the correctness of property A).
- **Data encapsulation** ensures that in those cases where the class does not provide any methods to modify a specific class or instance variable, only the correctness properties of the class itself need to be considered when reasoning about the possible values of such a variable. If the class does provide methods to modify the variable, then the pre- and postconditions of those methods must be taken into account and the designer has to ensure that the clients do not violate the preconditions when they invoke the methods. Data encapsulation therefore **restricts the number of correctness properties** that need to be considered during correctness arguments.
- The implications of using **inheritance** are manifold. Correctness properties of ancestor classes can be **reused as is** in the correctness arguments of properties of descendant classes. They can also be **overridden**, provided the **preconditions** are **not strengthened** and the **postconditions** are **not weakened**. New correctness properties may also be **added** in the descendant classes.
- Finally, it should not be underestimated how important it is to use correctness arguments to check **informally** that the **actual behaviour** of the system, as implied by the correctness properties of the **constituent** classes of the SLOOP program, matches the **intended behaviour** of the system, as implied by the specified correctness properties of the system. Correctness arguments are used to **confirm** the **choice of classes** selected to comprise the system.

As was stated earlier, the correctness reasoning during the design phase takes place at the level of the parallel statement, i.e. it is based on the atomicity of the parallel statement. The allocation of these statements to processors is not considered. The next chapter deals with the issue of

ensuring that the semantics of the SLOOP parallel statements are preserved when the statements are allocated to one or more processors, thereby ensuring that the correctness arguments used during the design phase are not invalidated by the mapping procedure.

CHAPTER 8

THE IMPLEMENTATION PHASE

8.1 Introduction

Up until now the target architecture of the system has been ignored, i.e. a unified approach is followed during the analysis and design phases. The implementation phase requires the consideration of a number of issues:

- The target architecture has to be determined.
- The objects and statements have to be assigned to processes/processors.
- The SLOOP program has to be mapped to an executable program.

The target architectures that are discussed in this chapter are:

- a sequential (von Neumann) architecture,
- a synchronous shared-memory architecture,
- an asynchronous shared-memory architecture and
- a distributed system.

The above list is not exhaustive, but was considered sufficiently distinct to demonstrate various types of mappings. These architectures are also discussed in [ChMi88] and it is therefore interesting to compare the mappings described in [ChMi88] with the mappings performed in the SLOOP method.

In Section 4.4.3 of Chapter 4 an overview of the mapping of a SLOOP program to an executable Smalltalk program was presented. It merely served as an introduction to the topic. This chapter elaborates on the following issues:

- Each type of target architecture requires a different type of mapping. The heuristics for the allocation of objects and SLOOP statements to processes / processors are given in Section 8.2.
- In Chapter 4 the derivation of an executable Smalltalk program for a sequential architecture was described. In Section 8.3 it is shown how the Smalltalk program can be adapted for other architectures, such as synchronous shared-memory, asynchronous shared-memory and distributed architectures.
- Further options regarding the mapping of the *macros-section* are discussed in Section 8.4.
- The mapping of more advanced types of SLOOP statements is described in Section 8.5. These descriptions cover the mappings of programs that contain multiple *quantified-statement-lists* as well as statements that contain multiple *statement-components* and *component-parts*.

- In Section 8.6 is shown how the reflective[1] facilities of Smalltalk can be used in order to make the mapping as transparent as possible to the class. Reflective computation can also be used to perform assertion checking.
- When the SLOOP program is mapped to a target architecture, it may be found that the level of parallelism displayed by the SLOOP program is not sufficient. This results in a return to the design phase. The SLOOP program is refined to introduce more parallelism, typically by decoupling the actions that appear in a single statement and by putting them into additional parallel statements. This topic is covered in Section 8.7.

At all times during the implementation phase the correctness properties specified during the analysis and design phases play an extremely important role. Additional correctness properties are defined for the infrastructures used during the implementation phase.

Note that the emphasis in this chapter is on Smalltalk as the target programming language. Where concurrency constructs are not required, the Smalltalk-80 language [GoRo89] suffices, otherwise the Concurrent Smalltalk language [Yoko90] is appropriate. However, the mappings discussed in this chapter are examples only. Many other mappings are possible, including mappings to other programming languages such as Java. As mentioned in Chapter 1, Smalltalk to Java translation has already been studied by other researchers [Enko98].

8.2 Mappings to various architectures

In order to map a SLOOP program to a target architecture, the SLOOP statements have to be **allocated to the process(es) / processor(s)** involved. The following four subsections discuss this allocation in the context of sequential, synchronous shared-memory, asynchronous shared-memory and distributed architectures respectively. Section 8.3 deals with the issues that need to be considered in order to ensure that the **semantics** of the SLOOP statements are retained during the mapping.

8.2.1 Sequential architectures

In the case of a sequential architecture, there is a single processor and a single process. Instructions are therefore executed strictly sequentially [Tane81]. When mapping a SLOOP program to such an architecture, all the statements are assigned to the same process on the same processor. Although the parallel statements are executed sequentially, their order of appearance is irrelevant. The only important issue is that they should be enclosed in an infinite loop in order to ensure that they are executed infinitely often. Each parallel statement should appear at least once within this loop.

8.2.2 Synchronous shared-memory architectures

Synchronous shared-memory architectures allow for multiple processors to share a common memory. There is a common clock and at each clock tick, each processor performs a single step of computation [ChMi88]. Multiple processors may read from the same memory location concurrently. If multiple processors write to the same location concurrently, they all have to write the same value. Concurrent read and write accesses to the same location are not allowed.

This type of architecture is particularly suited to take advantage of the **synchrony** inherent in SLOOP statements. Recall that the *component-parts* of a SLOOP statement (i.e. those parts separated by the '||' or '\+' symbols), execute in parallel. This means that each ***component-part*** of a specific SLOOP parallel statement can be assigned to a **separate processor**.

[1] The concept of computational reflection was described briefly in Chapter 1, Section 1.3.4. Further details are given in Section 8.6.

One statement is executed at a time, with each processor executing a *component-part* of that statement. Infinite loops are implemented on each processor in order to ensure that the *component-part*s of each statement will be executed inifinitely often.

8.2.3 Asynchronous shared-memory architectures

Asynchronous shared-memory architectures also allow for multiple processors to share a common memory. **Asynchronous** shared-memory architectures do **not** have a **common clock**, so the computation steps of the various processors might or might not execute simultaneously. If two processors access the same memory location simultaneously, the actual accesses occur in an arbitrary order [ChMi88].

Each parallel **statement** is assigned to **one** of the processors. (Multiple statements may be assigned to each processor.) If two statements do not send messages to the same object, they may execute concurrently, otherwise they have to execute in an arbitrary sequential order. This is to prevent **interference**[2].

In the case where a statement refers to shared objects, the sequential ordering is achieved in the following way: Before any statement may be executed, all the shared objects that are accessed by that statement have to be reserved (locked) by the processor to which the statement has been allocated. Since each statement may refer to **multiple** objects in the common address space, it would be possible to have a scenario where processor A has been granted a lock on object X and is waiting for a lock on object Y, while processor B has been granted a lock on object Y and is waiting for a lock on object X. Each processor will wait forever for the other to release the required lock. The two processors are therefore in a **deadlock**[3] situation.

One algorithm that guarantees the absence of deadlocks is to reserve the objects in a **prescribed order**. Issues such as performance also need to be considered when selecting an appropriate algorithm. This aspect will be discussed further in the next section, where mappings to distributed systems are described, since the same issues need to be considered when dealing with distributed systems.

In Section 8.3.3 a mapping to a specific type of asynchronous shared-memory architecture is discussed. It describes an architecture that consists of a **single processor**, but which has **multiple processes** running on it. The processes share a common address space. In that case each parallel statement is allocated to a specific process. In the mapping described in Section 8.3.3, no object reservation is required since there is only one processor, but it has to be guaranteed that each statement executes to completion before any other statement can start executing. That is to prevent interference. Although there is only a single processor, the execution of the statements allocated to the various processes can be interleaved in any arbitrary way, so in that sense the program fragments execute in parallel. One can therefore consider it an example of **pseudo-parallelism**.

In order to ensure that all the parallel statements are executed infinitely often when a SLOOP program is mapped to an asynchronous shared-memory architecture, each statement on each processor/process has to execute infinitely often. Thus, all parallel statements allocated to a processor/process have to be enclosed within an infinite loop on that processor/process.

---

[2] The concept of interference was defined in Chapter 2, Section 2.3.2.
[3] The conditions for deadlock were described in Chapter 4, Section 4.3.6.5.

8.2.4 Distributed systems

Distributed systems have multiple processors, each with its own local memory. Communication occurs via **message passing** [Bena90].

When a SLOOP program is mapped to a distributed architecture, each object referenced by the program is allocated to one of the processors. Multiple objects may be allocated to a single processor. Since the parallel and sequential methods of an object are executed on the processor where the object resides, messages have to be passed via some or other communication mechanism if a client invokes a method of a target object that is not co-located. The interface to the communication infrastructure will be discussed in more detail shortly.

At each processor all the parallel statements assigned to a specific process on that processor have to be enclosed in an infinite loop. This ensures that all the parallel statements of the program are executed infinitely often. One of the infinite loops at each processor also has to include a parallel statement that handles the messages received from remote objects. The purpose of this statement is to pass the messages to the relevant local objects.

All the shared objects that are referenced by a parallel statement (either directly from within the parallel statement itself, or indirectly via one of the methods invoked by the parallel statement), have to be **reserved** before the statement may execute. In Section 8.3.4.1 an algorithm for the reservation of resources in a distributed architecture is described. That particular algorithm was chosen for its **simplicity** rather than its **performance**. The objective in this chapter is to point out the issues that are involved, in which case a simple algorithm is the most appropriate.

When an object sends a message to another object and the latter is not co-located, a complex infrastructure is required in order to get the message to the target object. For example, the location of the target object has to be established and the message needs to be converted into a format that can be transmitted over a communication medium. Furthermore, the interface to the communication medium has to be handled. An infrastructure which provides such services is called a **middleware** infrastructure [OHE97]. One example is the Common Object Request Broker Architecture (CORBA) [OHE97].

In this chapter the emphasis is on how to ensure that the **semantics** of the statements of a SLOOP program are **retained** when the program is mapped to a distributed architecture. Once that mapping is done, any middleware infrastructure can be used to take care of the details of actually getting a message across to the target object, provided the selected infrastructure guarantees the reliable delivery of messages to the target object. Thus, the transfer of messages from one processor to another is transparent to the statements of the **mapped** SLOOP program.

8.2.5 Comparison with UNITY mappings

When comparing the UNITY mappings described in [ChMi88] with the SLOOP mappings discussed above, the most important difference lies in the fact that UNITY statements deal with **variables** (all UNITY statements are simple **multiple-assignment** statements), whereas SLOOP statements deal with **objects** and the **messages** sent to those objects. The semantics of a SLOOP statement are therefore more complex and that has to be taken into account when mapping a SLOOP program to a specific architecture. The next section describes in more detail the issues that need to be considered in order to ensure that the semantics of the SLOOP statements are retained during the mapping procedures. Although the object-oriented constructs add complexity to the SLOOP mapping procedures, this is offset by the **higher level of abstraction** that is achieved via the use of object-oriented concepts.

The object-oriented nature of the SLOOP method also gives it a distinct edge over UNITY when the program is mapped to a **distributed** architecture, since the SLOOP mapping can take advantage of middleware infrastructures such as CORBA. The UNITY mappings to distributed architectures are in terms of variables and are described in [ChMi88].

8.3 Deriving executable programs on various architectures

This section demonstrates how executable programs can be derived for the different architectures described in the previous section. The aim is to show what needs to be taken into account in order to ensure that the correctness properties of the system are not violated during the implementation phase.

8.3.1 Sequential architectures

The simplest type of mapping is to a sequential architecture. In Chapter 4, Section 4.4.3, it was shown how an executable Smalltalk program can be derived from a SLOOP program if the target architecture is sequential. Those parts of the executable program that differ for the various architectures are repeated here for the sake of convenience. In the sections that follow, the differences are highlighted.

The mapping of the *activation-section* of the `CallCentreSimulation` SLOOP program is shown below:

```
| aCC_SimulationActivation |

aCC_SimulationActivation :=
      CC_SimulationActivation setup.
[true] whileTrue: [aCC_SimulationActivation p_activate]
```

Thus, an instance of `CC_SimulationActivation` is created. In turn, it instantiates all the necessary classes. The program then enters an infinite loop. The latter contains a single statement which invokes the `p_activate` method of the `CC_SimulationActivation` class. At each invocation of the `p_activate` method one of its constituent parallel statements is executed.

The mapping of the `p_activate` method is now shown. In order to select only one statement at each invocation, while at the same time ensuring that each statement will be selected in turn, two additional instance variables are introduced to the class. The `p_activateTally` variable is set to the number of parallel statements that appear in the method and `p_activateCycleIndex` is initialised to `p_activateTally - 1`. At each invocation of `p_activate`, the `p_activate=CycleIndex` is incremented modulo the number of parallel statements in the method. Its value is then used to select the parallel statement to be executed.

For simplicity, only two of the statements of the `p_activate` method are shown in the example below.

```
p_activate
p_activateCycleIndex :=
      ((p_activateCycleIndex + 1) \\ p_activateTally).
"Determine which statement should be executed."

(p_activateCycleIndex = 0)
ifTrue: [timer p_runTimer: timerEventQ]          "statement 0"

ifFalse: [(p_activateCycleIndex = 1)
```

```
        ifTrue: [self p_categoriseAndAllocate]      "statement 1"

        ifFalse: [...
                 ]
        ]
```

The above example serves to illustrate another issue that needs to be addressed during the mapping procedure, viz. the handling of parallel messages to the pseudo-variable `self`. The statement containing a message to `self` may be mapped as shown in the example above, or it may be expanded, in which case the resulting statements are included in the mapping. The second option is given next.

```
p_activate
p_activateCycleIndex :=
      ((p_activateCycleIndex + 1) \\ p_activateTally).
"Determine which statement should be executed."

(p_activateCycleIndex = 0)
ifTrue: [timer p_runTimer: timerEventQ]          "statement 0"

ifFalse: [(p_activateCycleIndex = 1)
      ifTrue: [scAllocator p_categorise: inputQ
                  using: scContainer]            "statement 1"

      ifFalse: [ (p_activateCycleIndex = 2)
            ifTrue: [scAllocator p_allocate:
                        scContainer from: inputQ]
                                                 "statement 2"
            ifFalse: [...
                     ]
               ]
         ]
```

If the second option is used, the `p_activateTally` variable has to reflect the total number of statements after the expansion has taken place. The expansion is mandatory if the ALBEDO meta-object infrastructure is used to perform the parallel statement selection. This issue is explained in detail in Section 8.6.2.

A brief description of the functionality of each statement that is executed in the above methods is given in the corresponding SLOOP methods in Appendix B, Section B.2.

At this point it may seem that the SLOOP method introduces added complexity during the mapping phase, because additional variables and statements are required in the mapped program. However, in Section 8.6 it will be shown how the concept of **computational reflection** can be used to ensure that variables and statements that do not form part of the SLOOP class can be implemented in a metaclass.

8.3.2 Synchronous shared-memory architectures

As described in Section 8.2.2, each *component-part* of a parallel statement can be allocated to a separate processor. Execution of a SLOOP statement is performed as described in Chapter 4, Section 4.3.6.3. To recapitulate: All *if* clauses are evaluated first, followed by the evaluation of all message expressions representing arguments of other message expressions, as well as the evaluation of all message expressions that play the role of the receiver of a message. Only then are the assignments and/or outermost message expressions executed. If the *if* clause of any *conditional-component-part-list* evaluates to false, then no further computation is performed for the corresponding *component-parts*.

Execution is restricted to **one statement at a time**. If, for a specific statement, no *component-part* is assigned to a particular processor, then that processor is idle for the duration of that statement. Although other mappings to synchronous shared-memory architectures are possible (e.g. where processors are not left idle), the simplicity of this mapping makes it easier to reason about its correctness. Thus, even though the *component-part*s of a second statement might not reference any of the objects referenced by the first statement, the second statement is not executed while the first is still busy executing.

Each *component-part* on each processor has to execute infinitely often. Furthermore, it has to be ensured that the *component-part*s belonging to the same statement execute **simultaneously** on the relevant processors. As a result, all the processors comprising the system have to execute the statements of the SLOOP program in the **same order**. The *component-part*s allocated to a specific processor are therefore enclosed within an infinite loop in a specific order on that processor.

8.3.3 Asynchronous shared-memory architectures

This section describes **a special case of an asynchronous shared-memory architecture**, viz. one where **multiple processes run on a single processor**. The processes share a common address space. Whereas in the case of a **synchronous** shared-memory architecture the *component-part*s of a parallel statement are assigned to different processors, here **complete statements** are assigned to processes.

Note that a parallel statement that is assigned to a specific process may invoke other parallel or sequential methods. In that case the statements comprising those methods are also executed by the same process. A process may therefore contain parallel statements belonging to multiple objects. Typically, the parallel statements belonging to a specific object are all assigned to the same process. However, that is not always the case, as will be seen below when the mapping of the `p_activate` method of the `CC_SimulationActivation` class is discussed.

Only parallel statements are assigned to processes. The statements of a sequential method are executed by the process containing the parallel statement which invoked the sequential method.

The following is an example of how the SLOOP parallel statements of the call centre program can be assigned to multiple Smalltalk processes running on the same Smalltalk virtual machine. Any number of Smalltalk processes can be used. In this example 6 processes are created. The statement allocation is summarized as follows:

Process number	Statements
1	No parallel statements
2	Parallel statements that invoke the parallel methods of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes.
3	Parallel statements that invoke the parallel methods of the `TimerServices` class
4	Parallel statements that invoke the parallel methods of the `ServiceCategoryAllocator` class
5	Parallel statements that invoke the parallel methods of the `ServiceCategory` class
6	Parallel statements that invoke the parallel methods of the `Connection` class

Table 8-1. Statement allocation to Smalltalk processes.

Process 1 contains the mapping of the **sequential** statements of the *activation-section*. It instantiates the necessary classes via the `setup` method of the `CC_SimulationActivation` class. It also creates all the other processes.

The purpose of the **parallel** statements in the *activation-section* is to ensure that the necessary parallel methods of the various classes are invoked infinitely often. In the mapping to the **sequential architecture**, this is achieved by enclosing the statement that sends the `p_activate` message to the `CC_SimulationActivation` instance in an infinite loop.

In the mapping to the **asynchronous shared-memory architecture**, the parallel statements of the `p_activate` method need to be **spread** over the various processes. For example, the statement that invokes the parallel method of the `TimerServices` class appears in process 3 and the ones that invoke the parallel methods of the `ServiceCategoryAllocator` class are present in process 4. The `p_activate method`, which merely served as a convenient way to group all of these statements in the SLOOP program, is therefore not used in this mapping. The parallel statements contained within the `p_activate` method are executed directly from within the *activation-section*. Statements containing messages to `self` are also **expanded** before the allocation of objects to processors is made. The mapping of the parallel statements of the *activation-section* is therefore spread over the various processes. In each process these parallel statements are enclosed in an infinite loop.

The Smalltalk-80 statements of process 1 are as follows (for brevity only the statements for invoking the parallel methods of the `TimerServices` and `ServiceCategoryAllocator` classes are shown):

```
| aCC_SimulationActivation |

aCC_SimulationActivation :=
      CC_SimulationActivation setup.

[ [true] whileTrue:
      [...] ]fork.
      "Process 2: Invoking the parallel methods of the
      CommsProviderSimulator and ServiceProviderSimulator classes."

[ [true] whileTrue:
      [timer p_runTimer: timerEventQ] ]fork.
      "Process 3: Invoking the parallel methods of the TimerServices
      class."

[ [true] whileTrue:
      [scAllocator p_categorise: inputQ using: scContainer.
      scAllocator p_allocate: scContainer from: inputQ
      ] ]fork.
      "Process 4: Invoking the parallel methods of the
      ServiceCategoryAllocator class."

[ [true] whileTrue:
      [...] ]fork.
      "Process 5: Invoking the parallel methods of the ServiceCategory
      class."

[ [true] whileTrue:
      [...] ]fork.
      "Process 6: Invoking the parallel methods of the Connection
      class."
```

Discussion:

Since the address space is shared by all processes, the `CC_SimulationActivation` class can be used to create all the instances that need to be present after initialization (via its `setup` method). Thereafter the various processes are created. The statements that belong to a particular process are enclosed in a Smalltalk block[4] and the message `fork` is sent to the block. All processes in this example are created at the same priority level. As soon as the message `fork` has been sent to a block, the process associated with that block becomes part of the list of processes that are scheduled by `Processor`, the single instance of the Smalltalk library class `ProcessorScheduler`. The latter is responsible for scheduling processes in a Smalltalk-80 system [GoRo89].

Thus, in the example shown above, the `timer p_runTimer: timerEventQ` statement which invokes the `p_runTimer:` parallel method of the `TimerServices` class, is enclosed in a Smalltalk block. The process that was created when the message `fork` was sent to the block, will eventually be scheduled and at that time the `timer p_runTimer: timerEventQ` statement will be executed by that process.

It was stated in Chapter 4 that a parallel **method** always returns after the execution of **one** of its parallel statements. Its statements are executed infinitely often by virtue of the fact that the **method** is **invoked** infinitely often. In contrast, the parallel statements in the *activation-section* of the program are not enclosed in a method. They are executed infinitely often because that is what the semantics of a parallel statement in the *activation-section* imply. In the mapping to the Smalltalk environment, it is therefore desirable to restrict infinite loops to the parallel statements in the *activation-section*. The parallel methods are invoked infinitely often via the infinite loop(s) in the *activation-section*.

In the mapping to the sequential architecture there was only one infinite loop in the *activation-section*. For the mapping to the asynchonous shared-memory architecture, there is an infinite loop for each process.

As stated earlier in this section, a parallel statement may invoke other parallel or sequential methods. For example, the `timer p_runTimer: timerEventQ` statement invokes the `p_runTimer:` parallel method. This method contains three parallel statements[5], as can be seen below:

```
        currentTime := SmalltalkLibPkg:::Time now asSeconds
      [] lastTime := currentTime \+
        currentTick := (currentTick + 1) \\ (timeoutCollection size)
            if difference ≥ 1 and: [currentTimeoutElement isNil]
      [] timerEventQ addLast: currentTimeoutElement \+
        currentTimeoutElement updateEndTime \+
        currentTimeoutElement timerServicesCompleted: true \+
        (timeoutCollection at: readIndex) removeFirst
            if currentTimeoutElement notNil
```

The above method is called a leaf parallel method[6], because the statements it contains do not invoke any parallel methods. Since only one SLOOP parallel statement contained in a leaf parallel method should be executed at each invocation of the method, it is necessary to insert a

[4] A Smalltalk block (delimited by square brackets) is defined as "a description of a deferred sequence of actions" in [GoRo89]. If the message `fork` is sent to a block, then a new process is created containing the expressions enclosed by the block.
[5] Details of the p_runTimer: method of the TimerServices class are given in Appendix B, Section B.11.
[6] In Chapter 4, Section 4.3.5.4, a leaf parallel method is defined as a parallel method that contains parallel statements invoking sequential methods only.

`Processor yield`[7] statement after each statement **within** the leaf parallel method. The process therefore relinquishes control after the execution of each parallel statement, enabling the `Processor` to schedule another process running at the same priority level.

In order to guarantee that the mapping to the above asynchronous shared-memory architecture retains the semantics of the original SLOOP program, the following aspects therefore have to be checked:

First of all, **each parallel statement** of the SLOOP program has to be represented by a Smalltalk statement which is **executed infinitely often**. This is achieved by assigning the parallel statements of each object to a process, and enclosing the statements of each process in an infinite loop. The parallel statements of an object may be assigned to a process either explicitly or implicitly. In the above example the statement `timer p_runTimer: timerEventQ` of the `p_activate` method of the `CC_SimulationActivation` class was allocated **explicitly** to process 3. However, the statements of the `p_runTimer:` method of the `TimerServices` class were allocated **implicitly** to process 3. Thus, all parallel and sequential methods invoked by a parallel statement are implicitly allocated to the process containing the invoking parallel statement.

Secondly, it has to be ensured that each SLOOP parallel statement contained in a leaf parallel method executes **atomically**. Thus, such a statement has to complete its execution before another parallel statement may be executed. This is achieved by disallowing any Smalltalk methods that relinquish control in the **mapped** statements. Thus, the Smalltalk counterparts of the SLOOP statements may not contain any message expressions that would relinquish control to the `ProcessorScheduler` instance.

However, each mapped parallel statement in a **leaf** parallel method is **followed** by a `"Processor yield"` statement in order to ensure that no process will forever **prevent** any other processes of the same priority from running. That is also the reason why all the processes are created at the same priority. It must also be guaranteed that each statement within the infinite loop in each process will **terminate,** i.e. no infinite loops are allowed in any of the statements enclosed by the outermost infinite loop of the process.

Since only one parallel statement in a leaf parallel method can be executed at a time (there is only one processor and each parallel statement in a leaf parallel method executes atomically), it is not necessary to reserve any objects prior to the execution of a parallel statement.

The third issue that has to be checked is whether each parallel statement in each leaf parallel method will indeed get a turn to be executed. Since only one parallel statement is executed at each invocation of a parallel method, the mapping of such a method has to ensure that **each parallel statement** contained within that method **will eventually be executed**. One possiblility is to introduce auxilliary variables in order to keep track of which statement should be executed next. An example of the usage of such variables was given in Section 8.3.1.

8.3.4 Distributed systems

This section describes the issues that are at stake when a SLOOP program is mapped to a distributed system. In section 8.2.4 the **allocation of statements and objects** to processors in a distributed system was discussed. This section continues that discussion with the focus on **retaining the semantics** of the SLOOP statements when they are mapped to a distributed architecture.

---

[7] When the message `yield` is sent to the ProcessorScheduler instance, the latter is instructed to give other processes at the priority of the currently running process a chance to run [GoRo89].

The basic premise is that the **correctness properties** that hold for a SLOOP program, will also hold for its mapped executable program, provided the mapping is done in such as way that the mapped statements reflect the semantics of the SLOOP statements. In order to achieve this, several issues need to be addressed:

- The atomicity of the SLOOP parallel statements must be preserved,
- the evaluation order of the SLOOP statements must be preserved,
- the semantics of the SLOOP message expressions must be preserved and
- the computational model must be preserved.

In the discussion below, the emphasis is on the last bullet. This is because it provides an opportunity to demonstrate how possible solutions to some of the complexities of a mapping can be **reused**.

The issues that need to be considered in order to address the first three points are therefore only discussed briefly in this introduction. As far as the preservation of the atomicity of the SLOOP statements is concerned, the following aspects are relevant: First of all, a SLOOP parallel statement is never spread over the processes in the distributed system. It is always a complete statement that is assigned to a process. However, a parallel statement may send messages to objects that reside at other processes. If two parallel statements share objects, they may not execute simultaneously. The (arbitrary) ordering of the execution of such statements will be discussed further when the issues related to the computational model are discussed below.

The evaluation order that needs to be preserved for SLOOP statements was first given in Chapter 4, Section 4.3.6.3 and summarised in Section 8.3.2 of the present chapter. Since a complete statement is mapped to a process in a distributed architecture, there is nothing specific to a distributed architecture that needs to be noted in this regard.

The preservation of the semantics of the message expressions is affected more by the target programming language than by the target architecture. However, in the case of a distributed architecture one also needs to ensure that the infrastructure that is used to transfer messages between objects guarantees the delivery of those messages and that it also guarantees that they will be delivered in the correct order.

As stated above, the remainder of this section focuses on issues related to the SLOOP computational model. In order to ensure that each mapped parallel statement will execute infinitely often, they have to be enclosed in an infinite loop. Furthermore, it has to be guaranteed that a mapped statement will not forever prevent another statement from executing, i.e. each mapped statement will eventually terminate. No statement should therefore contain an infinite loop. The only infinite loop that is allowed, is the outermost loop in **each process** that ensures that each statement is executed infinitely often. If there are multiple processes per processor, then each mapped parallel statement should also be followed by a statement which will yield control to the process scheduler. Absence of deadlock[8] should also be guaranteed.

The remainder of this section covers various aspects related to the prevention of deadlock in a mapping to a distributed architecture. The first subsection focuses on the rules that need to be followed regarding the reservation of objects in order to ensure that deadlock will not occur. The second subsection deals with the identification of the objects that need to be reserved and the third subsection describes why the CORBA Concurrency Control Service [OHE97] is not used to handle the object reservation aspect of the mapping to distributed architectures.

[8] Conditions for deadlock and ways of preventing deadlock as presented in the literature were discussed in Chapter 4, Section 4.3.6.5.

8.3.4.1 Guaranteeing absence of deadlock in a mapping to a distributed architecture

As mentioned in the introduction, this description of the mapping of SLOOP programs to distributed architectures focuses on the role of the SLOOP computational model. In this section it is demonstrated how the SLOOP approach enables the system designer to work at a **high level of abstraction**. Recall that during the design phase, the system is designed in terms of a number of **atomic** parallel statements, each executing infinitely often. The designer may **rely** on this atomicity at the design level, thereby **simplifying** the correctness reasoning at that level. There is no need to be concerned with complicated mechanisms to ensure exclusive access to objects and to guarantee that critical sections are handled correctly. The designer merely includes all the actions that need to take place atomically in a single parallel statement.

When the SLOOP program is mapped to its target environment, this **atomicity** has to be **retained**. At the same time, the mapping also has to guarantee that each mapped parallel statement will execute infinitely often. Thus, no statement should ever prevent any other one from executing. Once an **infrastructure** has been developed which satisfies these requirements, it can simply be **reused** by each subsequent mapping. Note that this infrastructure is different from the middleware infrastructure discussed earlier. The latter provides services such as locating the various objects in the system and transforming the messages into a format that can be transmitted over a communication medium. The SLOOP infrastructure discussed here **uses** the services of a middleware infrastructure.

In the discussion that follows, the required SLOOP infrastructure comprises a system which controls the sequence in which parallel statements at various processors may execute. The infrastructure determines which objects are referenced as target objects by each statement, it requests exclusive use of those objects on behalf of each statement and implements an algorithm which determines in what order the exclusive access may be granted. A distributed resource allocation algorithm is used in the example below. The remainder of this section is devoted to a description of the functionality of such an infrastructure.

*Note: In the description that follows, it is assumed that the distributed system consists of multiple processors, with a **single process** running on each processor. At the end of this section the impact of having **multiple processes** per processor will be discussed.*

Issues to be considered:

In the architectures described earlier, it was only necessary to be concerned with the allocation of **statements** to processes/processors. The mapping of a SLOOP program to a distributed system **also** involves the allocation of **objects** to processors, since there is no shared memory amongst the processors. This means that the data of the object is stored in the private memory of the processor and its methods are executed by that processor. Since there are multiple processors that execute concurrently and an object may receive a message from one remote object while it is busy processing a message from another remote object, the issue of ensuring the **integrity** of objects has to be addressed.

The integrity of an object can be ensured if there is **no interference**. This can be achieved by requiring that **an object** only executes methods related to a single parallel statement at a time. The **sequential**[9] methods invoked by a parallel statement are invoked **synchronously**[Vino97]. Nested upcalls[10] are allowed, but only if the upcalls are related to the execution of the current

[9] Since parallel methods contain parallel statements and each parallel statement must execute infinitely often, the correctness properties that are defined for the parallel methods take into account that multiple parallel methods of that object will execute concurrently.
[10] Synchronous invocation and nested upcalls were defined and discussed in Chapter 3, Section 3.2.2.1.

(local or remote) parallel statement. If two parallel statements **share objects**, then only the methods related to **one** of these parallel statements may be executed at a time. This restriction ensures that the correctness arguments used during the design phase are preserved during the implementation phase. When a sequential method is executed, **the state of the object is defined by the total correctness property of the method being executed**. If the above restriction is adhered to, no other method of the object can interfere with the state of the object during that execution. In Chapter 4 this requirement was illustrated by two scenarions. In Figure 4-8(a) it was shown that the nested upcalls belonging to the same parallel statement would always result in the same execution sequence of the sequential statements, whereas the nested upcalls belonging to two different parallel statements sharing objects and executing simultaneously could result in arbitrary execution sequences.

However, by restricting execution at a specific processor to the methods belonging to a single parallel statement at a time, a new problem is introduced: it is possible for deadlocks to occur (a scenario for deadlock in such a situation is described in Chapter 3, Section 3.2.2.1), unless preventative measures are taken. In [Tane92] the reservation of resources is given as one possible mechanism to prevent deadlock. Thus, by ensuring that all the relevant objects are reserved for the exclusive use of a particular parallel statement prior to the commencement of the execution of that statement, deadlock can be prevented. The relevant objects are the object to which the parallel statement belongs, as well as all the target objects[11] referenced either explicitly (in the statement itself) or implicitly (when a message is sent to an object from within one of the methods that are invoked as part of the chain of methods that are executed by the statement).

Apart from guaranteeing the **integrity of a SLOOP object**, this solution also ensures that there is no **interference** when parallel statements are executed. A SLOOP parallel statement always contains a modifying part, optionally governed by a conditional part. The SLOOP model relies on the fact that the state of an object does not change between the time that the conditional part is evaluated until the modifying part is executed. While a parallel statement is being executed, no other parallel statement should interfere with the states of the objects referenced by the former.

Recall that the discussion in this section focuses on the case where **each processor** in the distributed system has only **one process** running on it. In the deadlock prevention strategy described below, it is assumed that each parallel statement **within a process** must execute to completion before the next parallel statement within that process can commence execution. Thus, if parallel statement *sA1* at processor A requires object *oA1* at processor A and object *oB1* at processor B, while parallel statement *sA2* within the **same** process requires object *oA2* at processor A and *oB2* at processor B, then the execution of statement *sA2* does not commence before the completion of statement *sA1* if the latter is currently being executed. This is because the atomic unit of execution **within a process** is a parallel statement. Although the parallel statements may be executed in an arbitrary order, each statement is executed to completion before the execution of the next statement is commenced.

This requirement is extended further to include the parallel statement which handles the messages received as a result of the execution of parallel statements located at remote processors. Thus, while the process at processor A is busy executing statement *sA1* and it is waiting for a response to its message sent to object *oB1* (which resides at processor B), the process at processor A is blocked except for nested upcalls belonging to the blocking statement. Thus, the process will only respond to messages received via the execution of statement *sA1*.

If **multiple processes** are implemented **at each processor**, the **efficiency** of the implementation can be improved considerably. In that case the parallel statement which receives messages from processes at other processors could be assigned to a separate process. It will then be allowed to

[11] A target object is an object to which a message is being sent. The client object is the object sending the message.

execute concurrently with the parallel statements in the other processes at that processor, provided the statements do not share objects. However, such improvements are not discussed here, since the purpose of this section is merely to describe one possible way of mapping a SLOOP program to a distributed architecture in order to be able to highlight the **reusable** aspects of the mapping.

The deadlock prevention strategies described below are therefore aimed at an architecture comprising multiple processors, but only one process per processor. They are based on the simple conceptual model that each parallel statement within a process executes to completion before any other parallel statement within that process is executed. This includes the parallel statement which handles messages from remote parallel statements.

This approach has the following implication: Even if two parallel statements at two different processors do not send messages to the same objects, deadlock could still occur if they send messages to objects that share **processors**. The scenario shown in Figure 8-1[12] supports this claim: Statement *sA1* at processor A requires objects *oA1* and *oB1*, while statement *sB1* at processor B requires objects *oA2* and *oB2*. Thus, there are no shared objects. However, the statements send messages **synchronously** to objects that share processors (*oA1* and *oA2* share processor A while *oB1* and *oB2* share processor B).

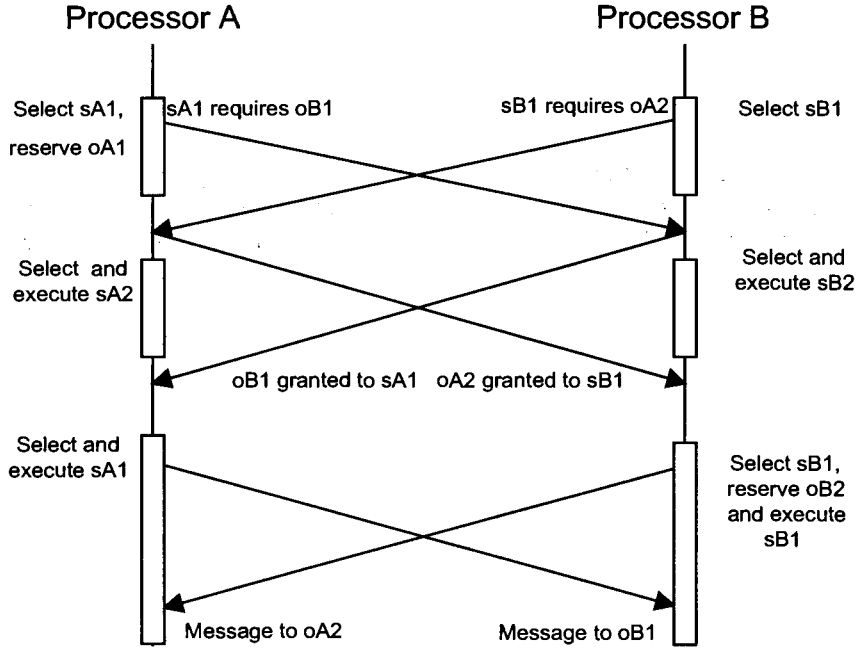

Figure 8-1. Scenario for deadlock when there are no shared objects, but the objects share processors in a single process per processor distributed architecture.

[12] The processors are identified via letters of the alphabet. The statements and objects at the various processors are identified by numbers prefixed by the letters designating the relevant processor, as well as an 's' to indicate a statement or an 'o' to indicate an object respectively. As is evident from Figure 8-1, reservation requests and confirmations are handled by each processor in between the execution of statements. Note that a statement is **selected** for execution on a round robin basis, but is only **executed** if all its reservation requests have been granted. At each processor one parallel statement is dedicated to the handling of messages from remote clients, i.e. it receives the messages from the clients and then passes them on to the relevant local objects. In Figure 8-1 this function is performed by statements *sA2* and *sB2*.

If the reservation requests are granted, deadlock occurs if the following happens: Statement *sA1* starts executing and sends a message to object *oB1* while at the same time statement *sB1* starts executing and sends a message to object *oA2*. Since the messages are being sent synchronously, the process at processor A blocks while waiting for the message to object *oB1* to complete, while the process at processor B blocks while waiting for the message to object *oA2* to complete.

The mapping of a SLOOP program to a distributed architecture where only **one process runs on each processor** therefore has to ensure the following:

- Each process only executes **one parallel statement at a time** and it executes it to **completion** before proceeding to the next one. Thus, parallel statements that are located at the same processor have to be executed in some (arbitrary) order.
- When two parallel statements are located at different processors and they **share objects**[13], the parallel statements have to be executed in **some (arbitrary) order**.
- Parallel statements that are located at different processors and that **do not share objects** (either explicitly or implicitly) have to be executed in **some arbitrary order** if the **target objects**[14] referenced by the different statements **share processors**.

Parallel statements may therefore execute simultaneously if the **statements are located at different processors,** they **do not share objects** (either explicitly or implicitly) and the **target objects** referenced by the respective statements **do not share processors**.

Addressing the issues:

One way of achieving all of the above is to acquire **all the resources** pertaining to a particular atomic execution **prior** to the commencement of that execution. This will automatically impose a sequential ordering on the statements that share resources. Based on the above discussion, it would appear as if these resources are the processors rather than the objects, because any statements that refer to target objects sharing the same processor may not execute simultaneously, even if these statements do not share any objects.

However, by viewing the processors as the resources that have to be reserved, the scope for concurrency becomes very limited. For that reason the target objects that are referenced by a statement are viewed as the resources, but special rules apply regarding the reservation of these resources. Details regarding these rules are given below.

The general strategy for object reservation:

Before providing detail regarding the resource reservation algorithm used here, the general strategy is described first. One of the aims of this algorithm is to ensure that no statement will be prevented forever from executing as a result of resource allocations to other objects, i.e. **each statement will eventually be granted its required resources and be allowed to execute**. Furthermore, while resources are being allocated to a statement, other statements will only be prevented from execution if their execution will result in the violation of the correctness properties of the system. Thus, **concurrent execution** must be **maximised** as far as possible.

---

[13] Two parallel statements share objects if they send messages to the same objects, or if the two parallel statements belong to the same object or if the one parallel statement sends a message to the object to which the other parallel statement belongs. Objects may be shared either explicitly via references in the parallel statement itself, or implicitly via references within the methods invoked by the parallel statement.

[14] If an object is not the receiver of the parallel statement under consideration, or it does not act as a target object during the execution of the parallel statement, but it is merely referenced (for example its value is assigned to a variable), then that object does not have to be considered when identifying the objects that could affect the possibility of deadlock. This is because none of the statements of such an object are executed during the execution of the parallel statement under consideration.

Requesting resources:

When a parallel statement is selected for execution, the location of each target object referenced by that statement is determined. A single reservation request is composed for **each** target processor. All the objects required from the specified processor for that particular parallel statement are listed in the reservation request destined for that particular processor.

The specified objects at the specified processor are allocated to the parallel statement in a single atomic action. There is therefore no need to reserve the **objects** located **at a particular processor** in a specific order. However, reservation requests are sent to **processors** in a prescribed order, since it cannot be guaranteed that all the resources pertaining to a specific statement will be granted simultaneously at all the processors involved. Ordering of requests removes the circular wait condition of deadlock [Tane92] and therefore prevents the object reservation algorithm itself from running into a deadlock situation.

For example, processor A has acquired objects from processor B and is now requesting objects from processor D, while processor C has acquired objects from processor D and is now requesting objects from processor B. Both A and C will wait forever for the outstanding resources as the resource allocation graph[15] [Tane92] in Figure 8-2(a) illustrates. If requests are always issued in some (arbitrary) order, deadlock is prevented, as shown in Figure 8-2(b). The request sent to the first processor in the sequence therefore has to be granted before the request to the second processor may be issued. In the example in Figure 8-2(b) processor C can only send its request to D once its request to processor B has been granted.

Note that if a statement sends messages to **local objects only and none of those objects have been allocated to or requested by other parallel statements, no reservation request is issued**. The statement may execute immediately, since it is guaranteed to complete and it cannot interfere with the execution of any other statement.

If a statement sends messages to **local objects only, but one of the objects has already been allocated to a remote parallel statement, a reservation request has to be issued.** This is to prevent interference. For example, if parallel statement *sB1* at processor B has been granted exclusive access to object *oA1* and it starts executing, it might check the state of object *oA1* in the conditional part of statement *sB1*. However, if statement *sA1* (which sends messages to object *oA1* only) is allowed to execute before statement *sB1* can execute its modifying part, it could change the state of *oA1*. This implies that the atomicity requirement of statement *sB1* would be violated. It is for this reason that statement *sA1* has to issue a reservation request if it is found that it has to send messages to objects that are currently allocated to or have been requested by a remote parallel statement.

If a statement sends messages to **both local and remote objects**, reservation requests are issued for the local objects as well as for the remote objects in the prescribed order.

[15] In a resource allocation graph processes are represented by circles and resources are shown as squares. If an arc is directed from a resource towards a process, then the resource has been granted to that process. If an arc is directed from a process towards a resource, then the process is waiting for that resource. A cycle in the graph indicates deadlock.

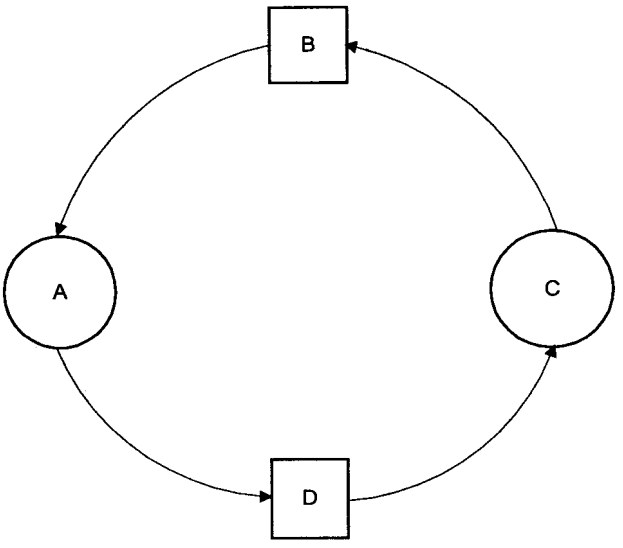

Figure 8-2(a). Resource allocation graph showing deadlock.

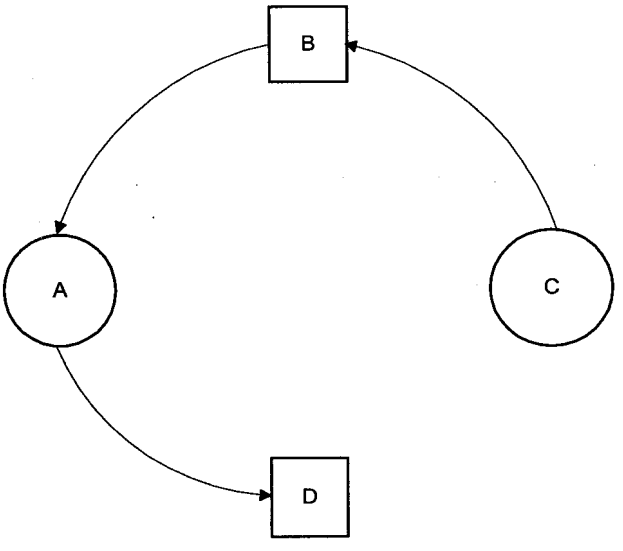

Figure 8-2(b). Deadlock is prevented if the resource allocation requests are made in an (arbitrary) order.

Granting resources:

When a request for the reservation of object(s) at a particular processor has been granted to a parallel statement (regardless of whether that statement is local or remote), all other reservation requests for objects at that processor are queued by the specified processor if the reservation requests are received from **remote** processors. The behaviour when the reservation request pertains to a local parallel statement will be described shortly. Once one or more of the objects at a processor have been allocated to a parallel statement, no other objects located at that processor are allocated to **remote** parallel statements until the resources have been released.

The rationale for this restriction is as follows: Once a resource has been granted to a remote processor, the parallel statement at the remote processor could start executing whenever that statement has acquired all its resources. If a local parallel statement has also acquired all its resources in the meantime, both statements could start executing simultaneously. This is because

the logic determining which statement can be executed next at a particular processor does not take the status at any remote processor into account. The only criterium that is used to decide whether a statement should start executing is whether all the required resources have been granted. As a result deadlock could occur if the parallel statements refer to target objects that share processors, since the method invocations are synchronous. This is demonstrated by the scenario depicted in Figure 8-3:

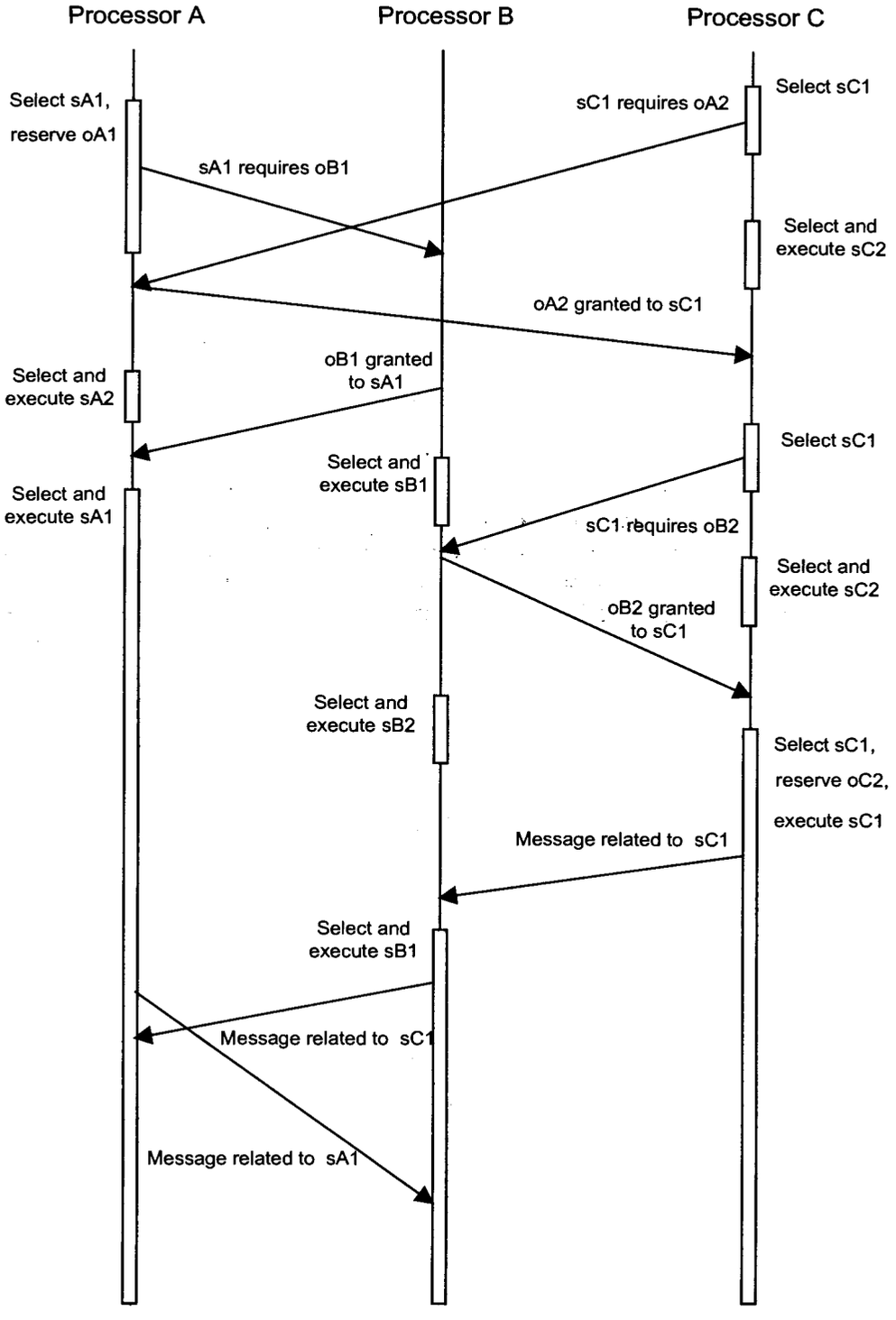

Figure 8-3. Deadlock as a result of the granting of multiple reservation requests.

Statement *sA1* at processor A requires objects *oA1* and *oB1* at processors A and B respectively. Statement *sC1* at processor C requires objects *oA2*, *oB2* and *oC2* at processors A, B and C respectively. All requests are granted, since there are no conflicts. Statement *sA1* starts executing. It sends a message to *oB1* and waits for a response. In the meantime statement *sC1* has also started executing. It has sent a message to *oB2*. Processor B is currently handling that message, which requires a message to be sent to *oA2*. Deadlock ensues, since processor A is still waiting for a response to the message sent to *oB1*.

If the restriction is adhered to, then the request for object *oA2* to be allocated to statement *sC1* is not granted by processor A. This is because a local object (*oA1*) has been reserved for a local statement that sends messages to both local and remote objects. As a result, when the request for object *oA2* is received, it is queued until statement *sA1* has released object *oA1*. Deadlock is therefore prevented.

In **more generic terms**, the above algorithm merely ensures that deadlock is prevented at the **processor level,** i.e. if two statements send messages to objects that share processors, then those statements are forced to execute in an (arbitrary) order. Thus, the circular wait condition for deadlock is eliminated [Tane92].

A **two-tiered approach** is therefore followed to prevent deadlock. **Requests** are made for exclusive access to the relevant **objects** for a particular statement. This allows statements to be executed in parallel if such concurrency will not compromise the correctness of the system, as will be seen below when the requests pertaining to local parallel statements are discussed.

On the other hand, when requests are **granted**, it is done at a per **processor** level if the request is received from a **remote** processor. If the reservation request pertains to a local parallel statement, more concurrency is possible, as will be seen shortly.

Note that even when a processor **grants a reservation request** to a remote parallel statement, it does **not** mean that the specified processor is **blocked**. The processor continues to execute its infinite loop. It will therefore continue to execute any local parallel statements that have been granted access to all the required objects. When a parallel statement that has reserved objects has **completed its execution**, all the objects that are reserved for that statement are **released** for allocation to other statements.

The treatment of **local reservation requests** is discussed next. Note that a local reservation request is only issued if a local parallel statement sends messages to both local and remote objects or if the local parallel statement sends messages to object(s) that have been allocated to or requested by another parallel statement. The reason for treating local reservation requests differently from remote reservation requests is as follows: If two local parallel statements do not share any resources, then the **efficiency** of the algorithm can be improved by allowing the resources for these statements to be **reserved** in parallel. Since these statements share the local processor, they will **not be executed** in parallel. The correctness of the system is therefore not compromised by the concurrent **reservation** of the resources required by these statements.

When a reservation request for a resource is issued for a local parallel statement, the request is queued if **any** local object has been allocated to a **remote** parallel statement or if any requests from remote parallel statement(s) have been queued. This is done to prevent a deadlock situation. If the scenario shown in Figure 8-3 is modified such that the request for object *oA2* reaches processor A before object *oA1* is reserved, then the request for object *oA2* will have been granted when the reservation request for object *oA1* is made. Thus a request for a local object is made when a local object has already been allocated to a remote parallel statement. It is clear that if the last request is granted, then exactly the same deadlock situation as depicted in Figure 8-3 is possible.

If no remote **statements** are involved, the request is granted if **all** the local objects listed in the request can be allocated, otherwise the request is queued. For example, if object *oA2* in Figure 8-4(a) has been allocated to statement *sA1* (i.e. to a local parallel statement sending messages to both local and remote objects) and statement *sA2* requires objects *oA2* and *oA3* (in addition to remote objects), the reservation request pertaining to statement *sA2* is queued, as shown in Figure 8-4(a). Object *oA3* is not allocated, even though it is free, since all the objects required by a specific parallel statement are allocated in a single atomic action. This ensures that whenever a statement releases its resources, then all the objects will be available to the next request in the queue.

If a reservation request is now issued for statement *sA3* indicating that object *oA3* is required (in addition to one or more remote objects), the request is also queued. Although object *oA3* is free, it has already been requested by statement *sA2* and should therefore be allocated to statement *sA2* first.

However, if the only local object required by statement *sA2* had been object *oA2*, then the reservation request for statement *sA3* would have been granted, as can be seen in Figure 8-4(b). The concurrent acquisition of the resources required by statements *sA1* and *sA3* is thereby facilitated.

Thus, the purpose of allowing multiple **local** reservation requests to be granted simultaneously, is to allow multiple local parallel statements that do not send messages to the same objects to **acquire their resources in parallel**. These statements cannot interfere with each other, since they are never processed in parallel (owing to the fact that they share the same processor). Note that it is only necessary to check whether the statements share **local** objects. If they share remote objects, the algorithm executed at the remote processors will ensure that the remote objects will not be allocated to both statements simultaneously.

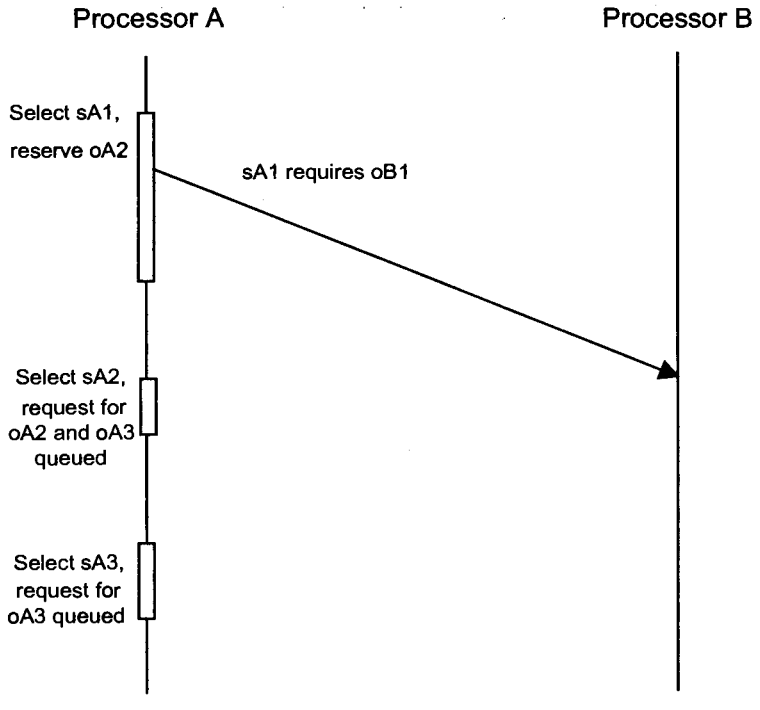

Figure 8-4(a). Handling of local reservation requests (shared local objects).

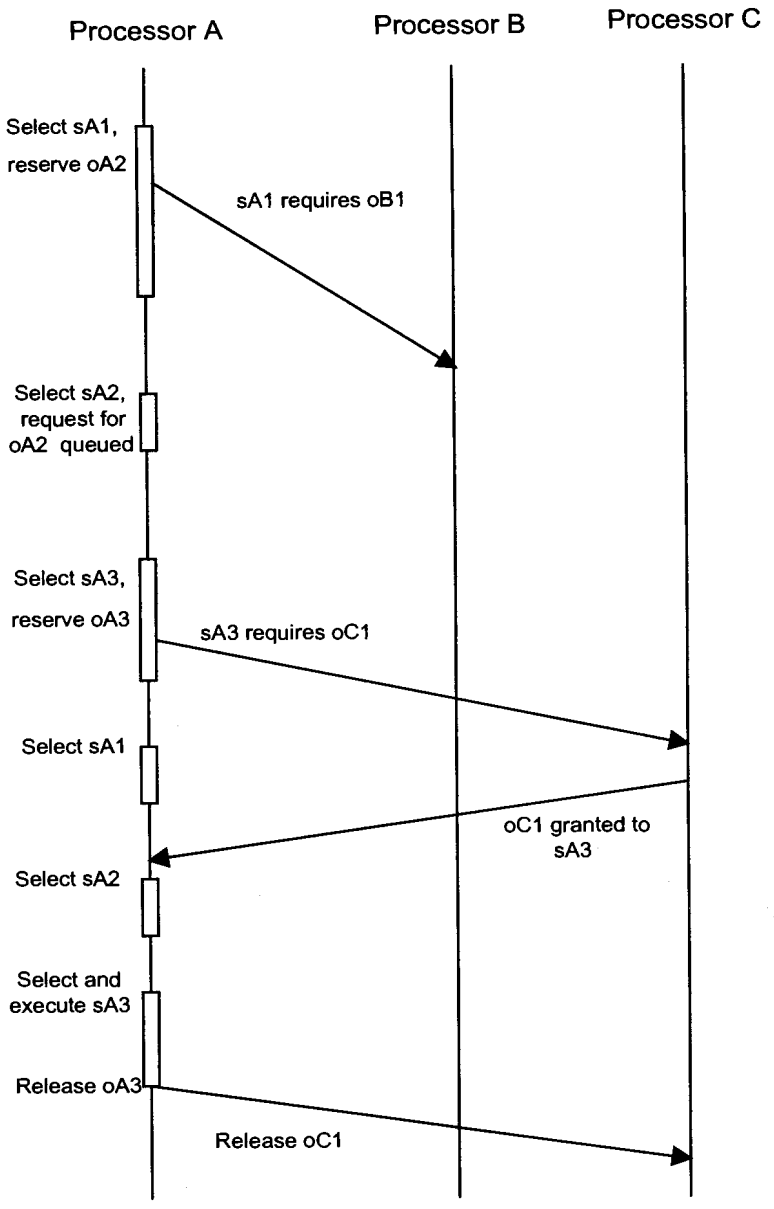

Figure 8-4(b). Handling of local reservation requests (no shared local objects).

It would be possible to achieve even more parallelism by allowing local parallel statements to obtain their remote resources concurrently regardless of whether they share local objects or not. This is because they will not execute simultaneously, since they share the local processor. However, that would complicate the resource allocation algorithm, since it would have to make provision for local objects being allocated to multiple local parallel statements at the same time. This option is not discussed further here, but this and other ways of improving the efficiency of the algorithm is a topic for further research.

The resource allocation algorithm:

At each processor in the distributed system, the following actions must be taken:

- Resource reservation requests received from remote processors must be handled.
- Each parallel statement in the infinite loop running on a processor must be selected infinitely often for execution. When a statement is selected for execution and all the relevant objects have already been allocated to it, it is executed. If there are still some objects outstanding, the necessary action is taken as described below. The statement is not executed.

- Indications that local objects have been released by remote parallel statements must be processed.
- Indications that remote objects have been allocated to local parallel statements must be processed.

A pseudo-code version of the basic functionality of the resource allocation algorithm is now presented. For easy reference, the different sections of the algorithm are referred to as rules and are numbered.

```
[
   (reservation request received from remote processor)                "Rule 1"
   ifTrue:
   [
      (any local object(s) already allocated to or requested by a
       local or remote parallel statement)
         ifTrue: [append the request to the local Request Queue
                  if it is not already present in the local Request Queue]
         ifFalse: [grant the request to the remote parallel statement]
   ]
   ifFalse:
   [
      (local objects released)                                         "Rule 2"
      ifTrue:
      [
         (all the local objects required by the first entry in the
          local Request Queue available)
         ifTrue:
         [
            grant the request to the entry in the local Request Queue.
            (the request is for a local parallel statement)
            ifTrue:
            [
               (all objects now reserved)
               ifTrue: [execute the statement and then release all
                        objects held by the statement]
               ifFalse: [issue the next request to a remote processor]
            ]
            ifFalse: [no further action taken]
         ]
         ifFalse: [no action taken]
      ]
      ifFalse:
      [
           (request granted at a remote processor)                     "Rule 3"
           ifTrue:
           [
              (access granted to all remote objects that should be reserved
               prior to the local objects)
              ifTrue:
              [
                 (any of the required local object(s) already allocated to
                  or requested by another local or remote parallel
                  statement)
                 ifTrue: [create a request and queue it in the local
                          Request Queue if it is not already present in the
                          local Request Queue]
                          "to prevent deadlock and interference"
                 ifFalse:
                 [
                    allocate the objects to the local parallel statement.
                    (all objects now reserved)
                    ifTrue: [execute the statement and then release all
                             objects held by the statement]
                    ifFalse: [issue the next request to a remote processor]
                 ]
              ]
```

```
                ifFalse: [issue the next request to a remote processor]
            ]
            ifFalse:
            [
                "next local parallel statement selected"
                (next local parallel statement requires local objects only)
                                                                    "Rule 4"
                ifTrue:
                [
                    (any of the required local object(s) already allocated to or
                     requested by a remote parallel statement)
                    ifTrue: [create a request and queue it in the local
                            Request Queue if it is not already present in the
                            local Request Queue]
                            "to prevent deadlock and interference"
                    ifFalse: [allocate the objects to the local parallel
                             statement and execute the statement]
                ]
                ifFalse:
                "next local parallel statement requires local and remote objects"
                                                                    "Rule 5"
                [
                    (access granted to all remote objects that should be reserved
                     prior to the local objects)
                    ifTrue:
                    [
                        (any of the local objects already allocated to or requested
                         by a remote parallel statement or
                         any of the required local object(s) already allocated to
                         or requested by another local parallel statement)
                        ifTrue: [create a request and queue it in the local
                                 Request Queue if it is not already present in the
                                 local Request Queue]
                                 "to prevent deadlock and interference"
                        ifFalse:
                        [
                            allocate the objects to the local parallel statement.
                            (all objects now reserved)
                            ifTrue: [execute the statement and then release all
                                 objects held by the statement]
                            ifFalse: [issue the next request to a remote processor]
                        ]
                    ]
                    ifFalse:
                    [
                        issue the next request to a remote processor
                    ]
                ]
            ]
        ]
    ]
]
```

As is evident from the above, the algorithm implements five rules:

Rule 1:
When **a reservation request** is received from a **remote** processor (the requesting processor), the request is **queued**[16] in the local Request Queue instead of granted if local object(s) are already allocated to or requested by a parallel statement. The latter may be a local parallel statement or one at another processor. It may even be another parallel statement at the requesting processor.

[16] A request is only queued if it is not already present in the local Request Queue. This applies to all the rules listed here.

Rule 2:
When **local** objects are **released** by a local or remote parallel statement, the next entry in the local Request Queue is inspected.

- If this entry is a request for local object(s) to be allocated to a parallel statement at a **remote** processor, the request is granted if all local objects are available. Otherwise the request remains in the queue until the remaining objects have also been released. This takes care of the requests queued as a result of Rule 1.
- **Else** if the entry is a request for local object(s) to be allocated to a **local** parallel statement, the request is granted if the requested objects are now free. This ensures that requests queued as a result of Rule 4 or 5 are processed. If all the required objects are now reserved, the statement is executed and then the objects are released. If all the required objects have not been reserved yet, the request to the next processor in the sequence is issued.

Rule 3:
When an indication is received that **remote objects have been reserved** for a local parallel statement, the algorithm checks whether there are any more objects that need to be reserved.

- If all objects have not been reserved yet and the next request in the sequence is one requesting remote objects, a request is sent to the relevant remote processor.
- **Else** if local objects should be reserved next, a request is **created and queued** in the local Request Queue if **any** of the required local objects are currently allocated to or requested by a **remote** parallel statement. It is also queued if the **specified** local object(s) are currently allocated to or have been requested by a **local** parallel statement.
- **Else** if all the requests pertaining to this statement have been granted, the statement is executed and then the objects are released.

Rule 4:
When the **next local parallel statement has been selected for execution** and all the **required objects** are **local**, a request is **created and queued** in the local Request Queue if **any** of the required local objects are currently allocated to or requested by a **remote** parallel statement. If all the required local objects are available, the statement is executed.

Rule 5:
When the **next local parallel statement has been selected for execution** and **both local and remote objects** are required, the following actions are taken:

- A request is sent to a remote processor if all objects have not been reserved yet and the next request in the sequence is one requesting remote objects.
- If local objects should be reserved next, a request is **created and queued** in the local Request Queue if **any** of the required local objects are currently allocated to or requested by a **remote** parallel statement. It is also queued if the **specified** local object(s) are currently allocated to or have been requested by a **local** parallel statement.
- **Else** if all the requests pertaining to this statement have been granted, the statement is executed and then the objects are released.

Thus, the **underlying goal** of this algorithm is to ensure that each statement in the infinite loop on each processor will always **terminate**[17], thereby ensuring that each parallel statement can be executed infinitely often. If all the local and remote resources required by a parallel statement have been granted to it according to the procedures specified above before it starts executing, its termination is guaranteed. **Separate** reservation requests are made for **each parallel statement**, since the latter is the atomic unit of execution.

[17] It is assumed that all the sequential methods invoked by the parallel statements are terminating methods.

Additional comments on the resource allocation algorithm:

- The entries of the local Request Queue are always processed on a strictly First In First Out basis.
- The processors are ordered in some (arbitrary) order and reservation requests for each parallel statement are issued strictly in this order.
- All requests for resources are always issued at the processor where the parallel statement is located. When a message is received from a remote object and the target object on the local processor needs to send a message to a remote object on yet another processor, it is therefore guaranteed that the execution will terminate, since all the resources required by the parallel statement on the originating processor have to be allocated to that statement before it starts executing.
- All resource requests are issued asynchronously, i.e. control is returned to the requesting processor without waiting for a response. Before any parallel statement is selected, the queue containing requests and responses from remote processors is examined. If it is not empty, its entries are processed.

- When multiple processors are involved, the sequential statements appearing in the *activation-section* of the SLOOP program are allocated to the various processors as appropriate. All of these sequential statements have to complete execution before the parallel statements start executing. One mechanism to achieve the desired behaviour is to prohibit the granting of object(s) to a remote **parallel** statement if all the sequential statements in the *activation-section* allocated to the specified processor have not yet completed execution. The requests are queued at the target processor. Note that the objects may be granted to a **sequential** statement appearing in the *activation-section* on a remote processor.

- If the order in which the sequential statements in the *activation-section* execute is significant, then the necessary synchronization mechanisms have to be included for this purpose during the mapping procedure.

Scenarios to illustrate aspects of the resource allocation algorithm:

Several scenarios are now presented to elucidate the above principles. Figure 8-5 illustrates how reservation requests are queued when the requested objects are not available. It also shows that reservation requests for a specific parallel statement are made sequentially. Only one reservation request may be outstanding at a time for a particular parallel statement.

There are three parallel statements at processor A. Statement *sA1* is selected for execution. It requires objects *oA1*, *oB1* and *oC1*. Object *oA1* is allocated to statement *sA1*. A request for object *oB1* is issued. Without waiting for a response from processor B, statement *sA2* is selected. It requires objects *oA2*, *oB1* and *oD1*. Object *oA2* is allocated to statement *sA2*. Object *oD1* is available. However, no request for object *oD1* is made, since it has not yet acquired object *oB1*. A request for object *oB1* is issued for statement *sA2* and queued at processor B. The third parallel statement (*sA3*) services messages that are received from remote objects, therefore it does not need to reserve any objects.

By the time that statement *sA1* is selected again, object *oB1* has been granted to it, therefore a request for object *oC1* is issued. The latter cannot be allocated to statement *sA1* yet, so the request is queued at processor C. Statement *sA1* is not executed, since an object request is still outstanding. Statement *sA2* is now selected for execution. No action is taken, since it is still waiting for object *oB1* to be granted to it.

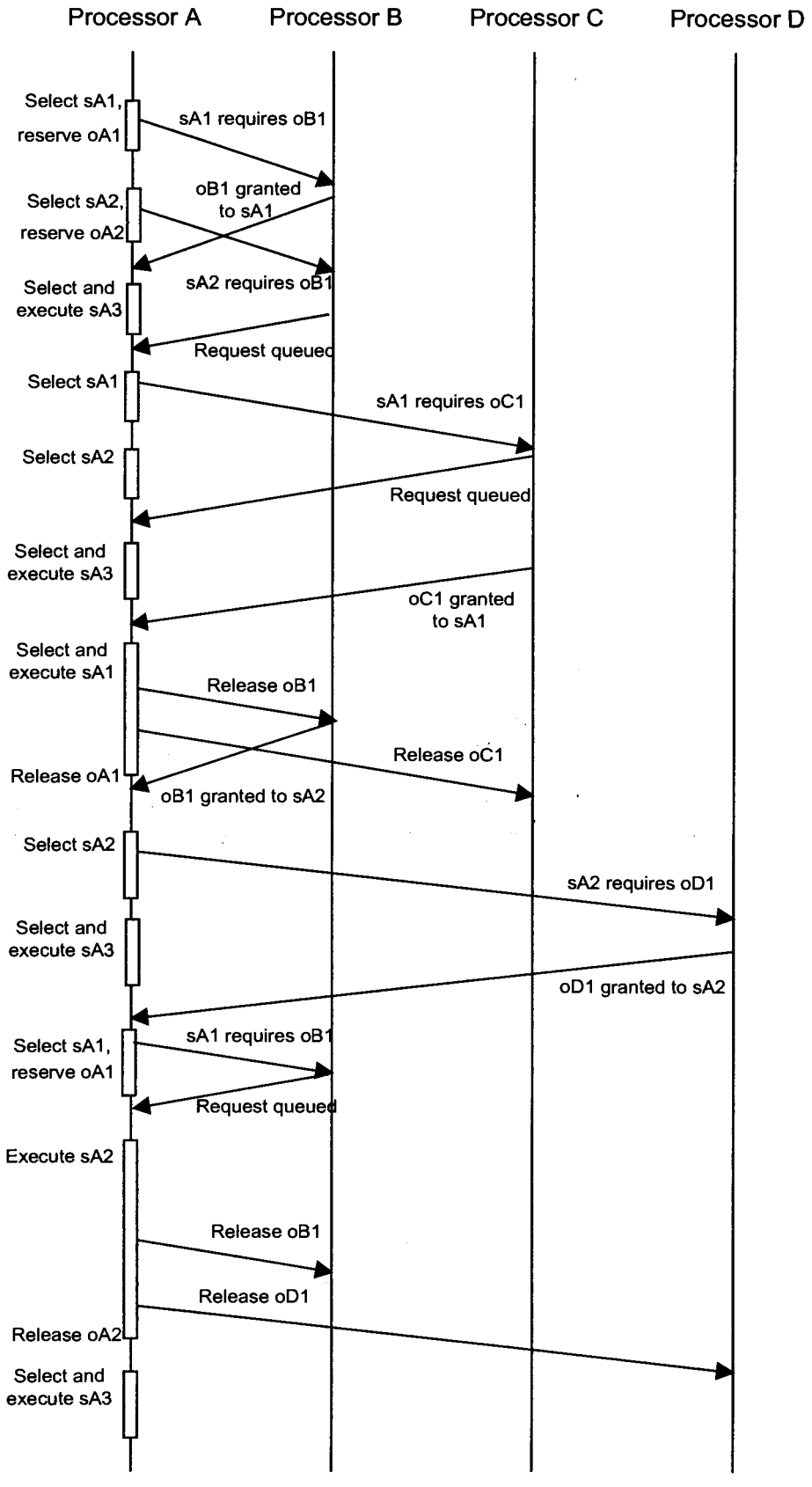

Figure 8-5. Scenario illustrating how resources may be requested by and granted to parallel statements at processor A.

Once the request for object *oC1* has been granted, statement *sA1* is executed. (The messages sent to the objects located at processors B and C are not shown in Figure 8-5.) When it has completed, it releases all its resources. The request for object *oB1* can now be granted to statement *sA2*. A request for object *oD1* is made on behalf of statement *sA2*. The latter only executes once it has acquired all its resources.

The next scenario illustrates how the resource allocation algorithm presented in this section allows for concurrency without introducing undue complexity. While local objects are allocated to parallel statements that require objects at multiple processors, the local processor may still execute local parallel statements that are unaffected by these object allocations. In particular, it may execute:

- the local parallel statement which services messages from remote objects,
- local parallel statements that do not send messages to remote objects and which do not require the local objects that are currently allocated to or requested by remote parallel statements, as well as
- local parallel statements to which all resources have been granted.

In the scenario depicted in Figure 8-6 processor B has two local parallel statements. The first one (statement *sB1*) only sends messages to objects *oB2* and *oB3* at processor B and the second one (statement *sB2*) services messages from remote objects. While object *oB1* is allocated to statement *sA1*, processor B is allowed to execute its local parallel statements that do not require any remote resources and that also do not require object *oB1*, as shown in Figure 8-6.

If the request for object *oC1* is granted and processor A starts executing statement *sA1*, a message related to statement *sA1* at processor A may arrive while processor B is executing statement *sB1*. Since the latter only sends messages to objects *oB2* and *oB3*, local objects that are currently not allocated to other statements, it is guaranteed to terminate. Processor B will therefore eventually execute its second parallel statement (*sB2*), which services messages from remote objects. That implies that the message from statement *sA1* will eventually be serviced.

Note that the statement which services messages from remote objects is always executed when it is selected, since no resources need to be reserved in order to execute this statement. By the time a message is received from a remote object, all the resources related to the statement to which this message belongs have already been reserved. For example, in the above scenario all resources required for the parallel statements on processor A are issued at that processor. Other processors never need to issue requests for resources on behalf of processor A. By the time statement *sA1* is allowed to execute, objects *oB1* and *oC1* are already allocated to it.

Thus, when processor B executes statement *sB2* (which results in the processing of the message related to *sA1*), it does not have to reserve any resources prior to the execution of the message. Even if the processing of this message results in a message being sent to an object at processor C, it is guaranteed that object *oC1* will be available to statement *sA1*, since it has been reserved for that statement by processor A.

If there is a fourth parallel statement (*sA4*) on processor A which only refers to objects *oA3* and *oA4* at processor A, that statement may also execute while statements *sA1* and *sA2* have not yet acquired all their resources. This is allowed, since a parallel statement which only refers to local objects that are not allocated to other remote statements at that moment, will never interfere with the state of an object that is allocated to another object. The statement will also always terminate, allowing the processor to execute the next statement in its infinite loop.

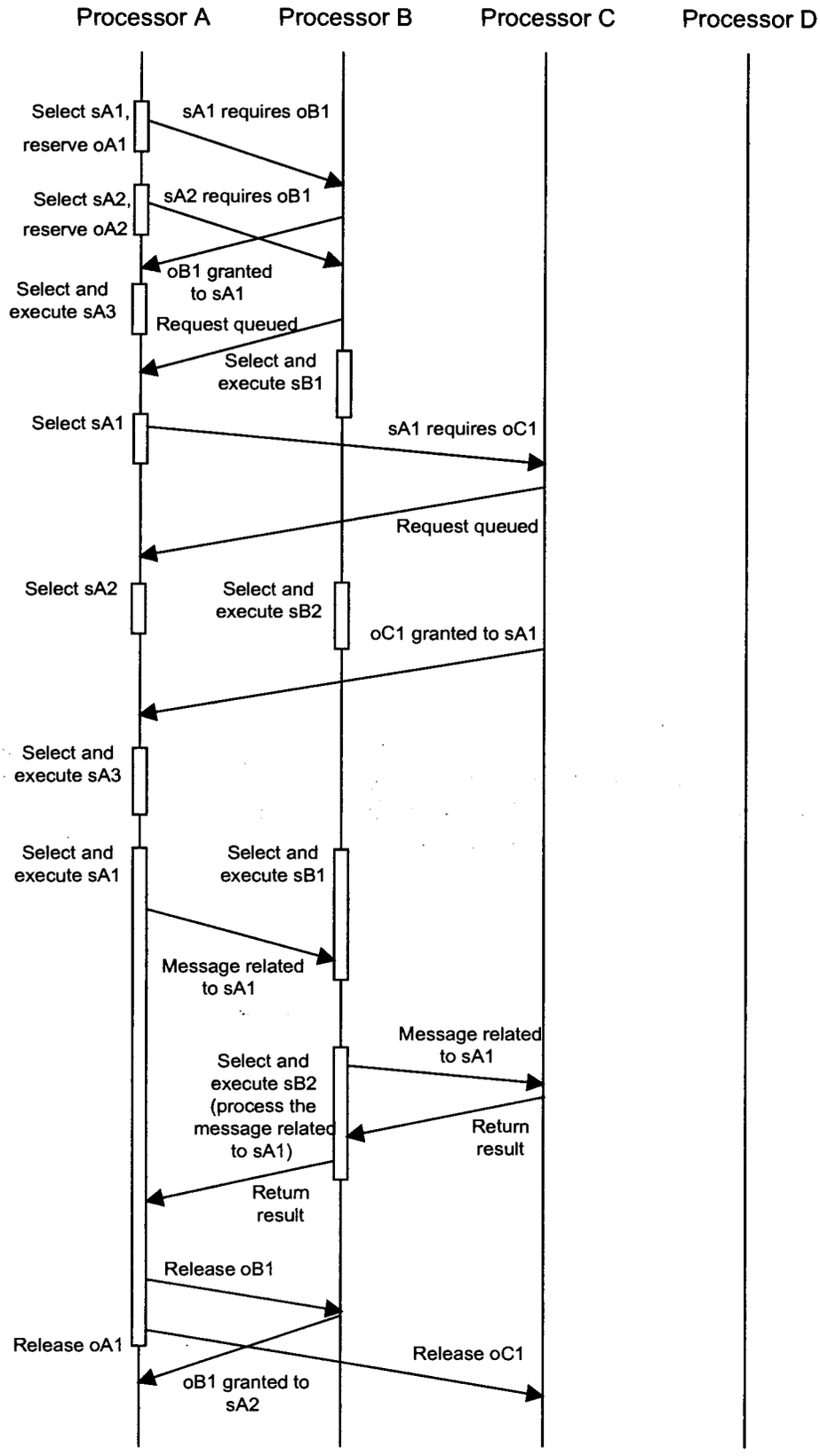

Figure 8-6. Scenario illustrating that a local parallel statement is always allowed to execute if it does not send messages to remote objects and also not to local objects that have been allocated to or requested by a remote parallel statement.

The scenario in Figure 8-7 illustrates the third condition under which a local parallel statement may be executed while some local objects are allocated to other parallel statements. In this case processor B contains a third parallel statement (*sB3*) which requires objects *oB3*, *oD1* and *oE1* and a fourth one, which requires objects *oB4* and *oC1*. When statement *sB3* is selected for execution, object *oB3* is allocated to it. A reservation request is sent to processor D in order to acquire object *oD1*. Statement *sB4* is executed next. A local reservation request is issued for object *oB4*. The request is granted, since the object is available and no local objects have been allocated to **remote** parallel statements. A request for object *oC1* is issued.

The requests for objects *oD1* and *oC1* are granted. When a reservation request for object *oB1* is received from processor A, the request is queued, because a reservation request from a remote parallel statement is always queued if any local objects are already allocated at that time. When statement *sB3* is selected again, it issues the request for object *oE1*.

Statement *sB4* is executed when it is selected, since it has acquired all its resources. It is not affected by the resource allocation status of statement *sB3*. Even if the request for *oE1* is received before statement *sB4* starts executing, statement *sB3* cannot interfere with statement *sB4*, owing to the fact that only one parallel statement is executed at a time at a particular processor and that statement execution is completed before the next statement is selected. (The messages sent to the objects located at processors C, D and E are not shown in Figure 8-7.) Once a parallel statement has completed execution, its resources are released.

The release of object *sB3* results in the allocation of object *oB1* to statement *sA1*. When statement *sB4* is selected, the local reservation request for object *oB4* is queued due to the fact that a local object (*oB1*) is allocated to a **remote** parallel statement at that time.

Figures 8-8(a) and (b) illustrate some of the consequences if these rules as implemented in the resource allocation algorithm presented in this section are not adhered to. Deadlock as a result of reservation request collision is shown in Figure 8-8(a). Processor A allocates object *oA2* to statement *sB1*, even though a local object is already allocated to statement *sA1*. This violates Rule 1. Processor B violates Rule 5 when it allocates object *oB2* to statement *sB1*. Deadlock ensues when statements *sA1* and *sB1* both start executing simultaneously.

In Figure 8-8(b) Rule 5 is violated at both processors, eventually resulting in deadlock. Figures 8-9(a) and (b) show how the rules listed above ensure correct operation under similar circumstances.

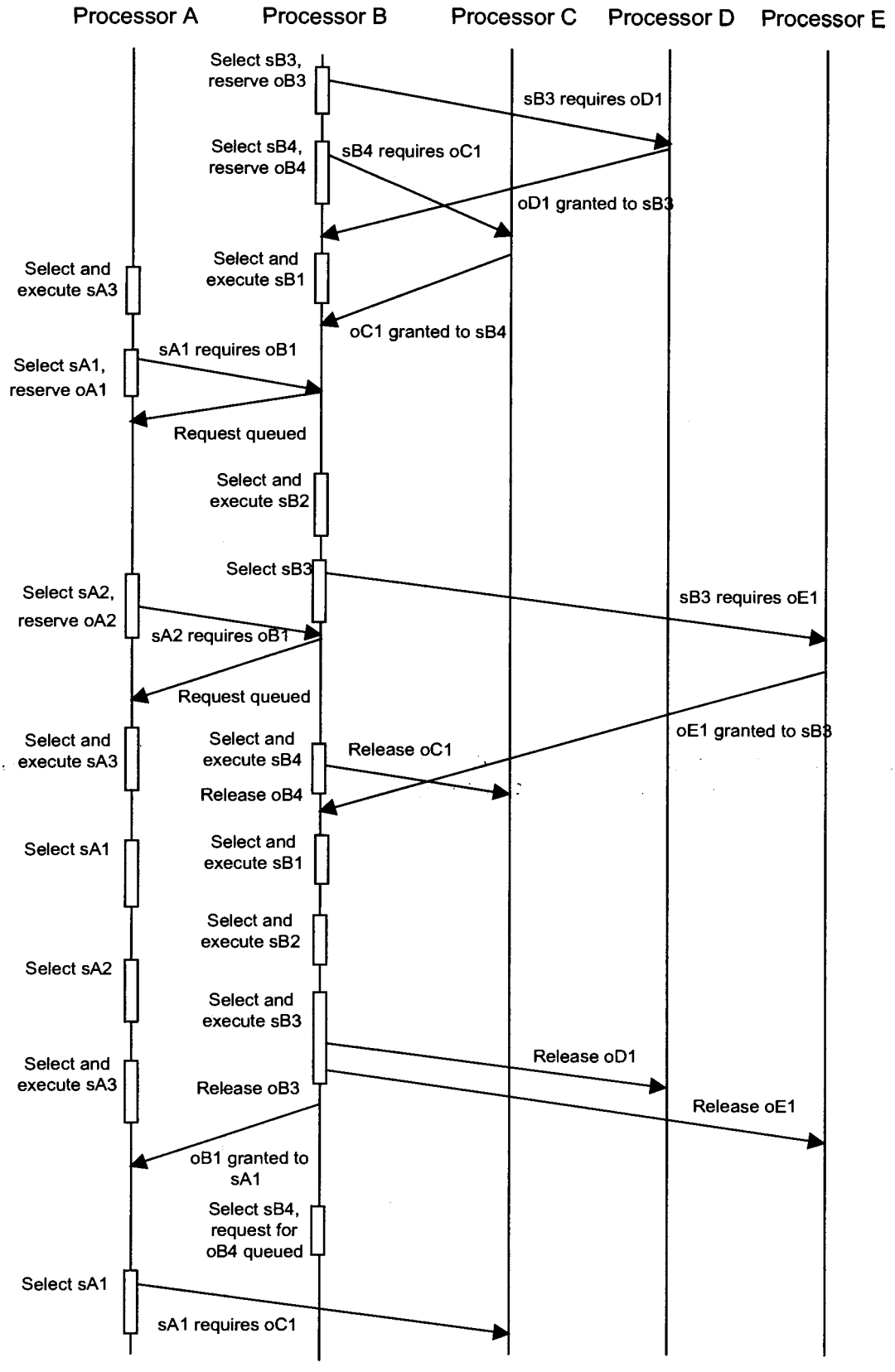

Figure 8-7. Scenario illustrating that a local parallel statement for which all resources have been granted may be executed while another local parallel statement is still waiting for remote resources to be granted. The two statements do not share local objects.

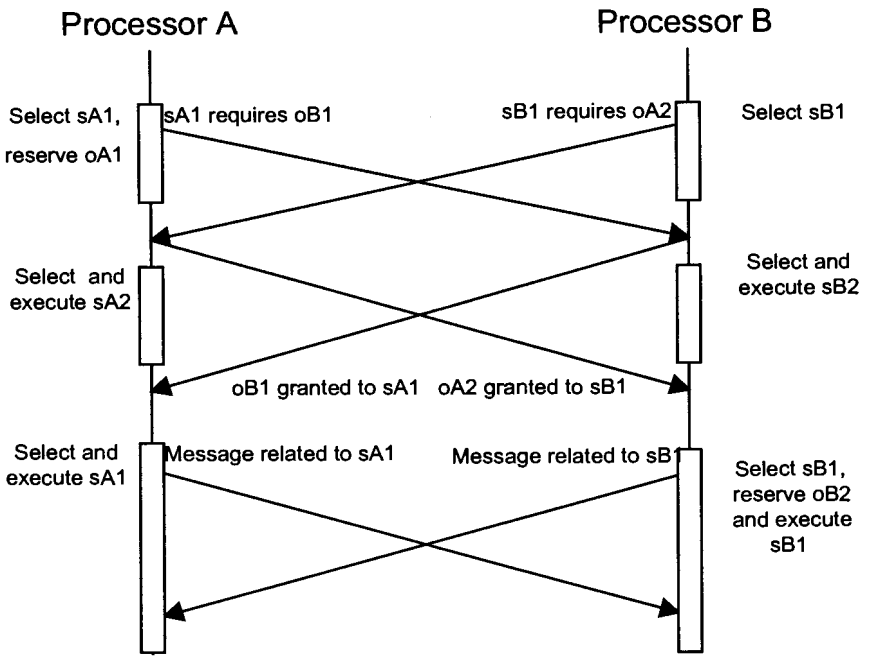

Figure 8-8(a). Deadlock in the case of reservation request collision when both local and remote reservation requests are granted at each processor.

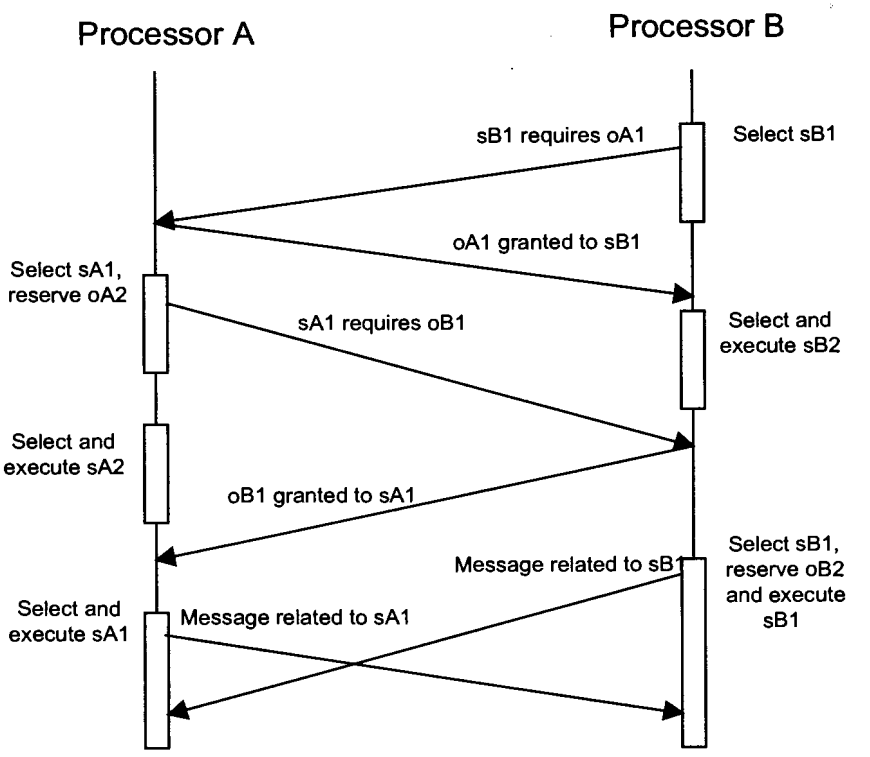

Figure 8-8(b). Deadlock when a local reservation request is granted while a local object is currently allocated to a remote parallel statement.

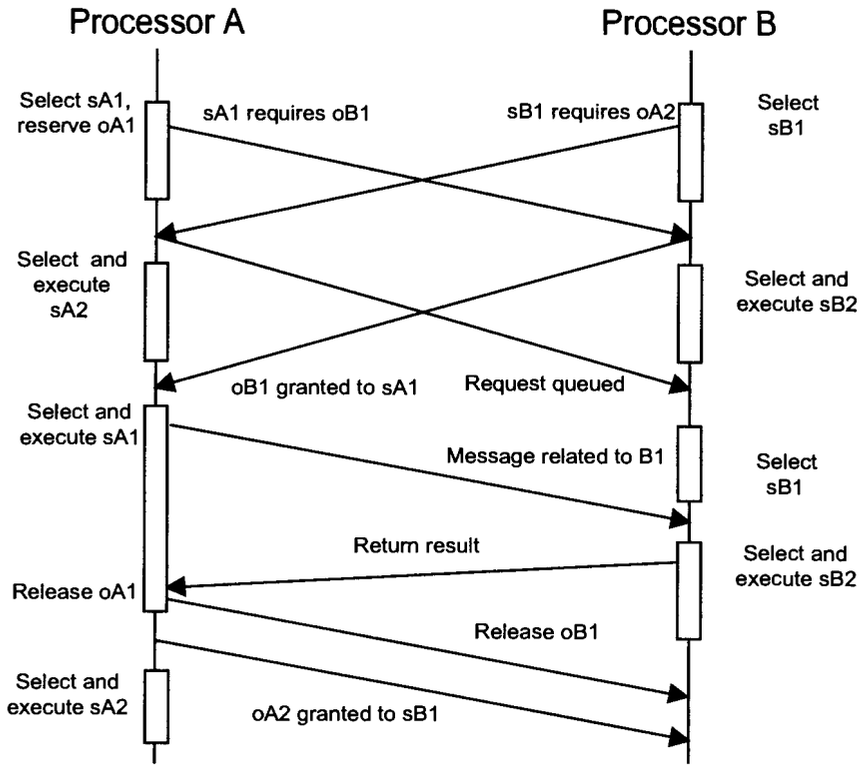

Figure 8-9(a). Deadlock prevention in the case of reservation request collision.

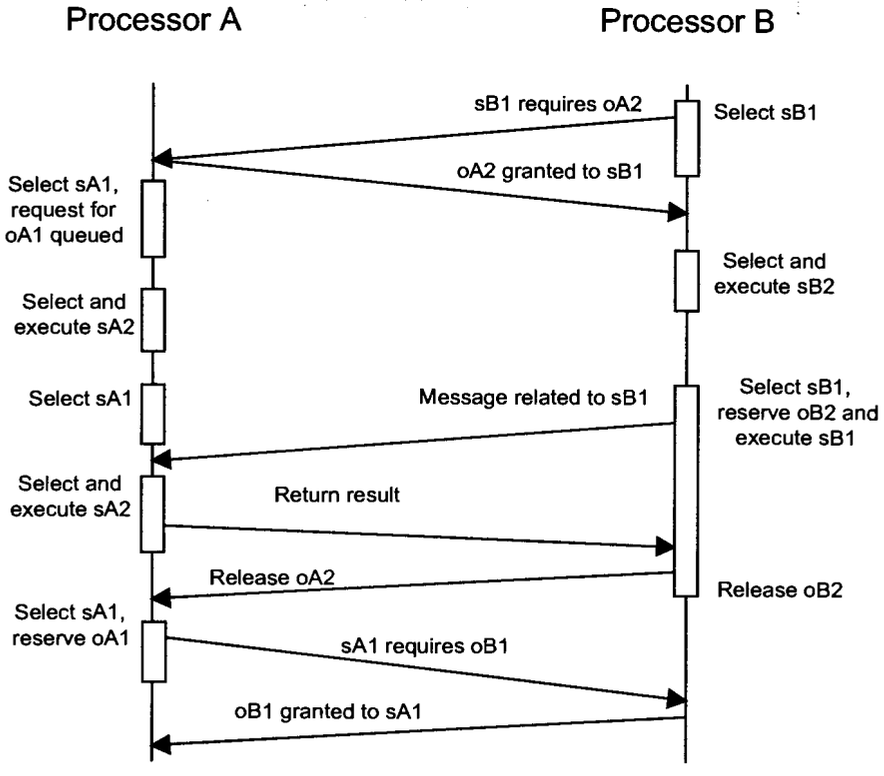

Figure 8-9(b). Deadlock prevention because a reservation request for a local object is queued while a local object has been allocated to a statement at a remote processor.

The effects of having multiple processes running on each processor:

If a distributed system comprises multiple processors, where each processor has **multiple processes** running on it, the effects on the resource allocation algorithm as described in this section are as follows:

In such a system there are multiple infinite loops running on each processor, one for each process. Each process relinquishes control of the processor in the same way as described in Section 8.3.3, i.e. the mapped SLOOP statements may not contain any messages that will yield control to the process scheduler, but after each parallel statement has been executed, control is relinquished explicitly. The atomicity of each parallel statement is thereby guaranteed.

If multiple processes may be present at each processor, only one of the processes at each processor would need to contain the statement which handles messages from remote objects. This is because the objects at that processor are shared by all the processes. It is only the parallel statements that are assigned to individual processes. In order to ensure the integrity of objects, parallel statements that share objects may not execute simultaneously.

Earlier in this section it was described how the efficiency of the resource allocation algorithm could be improved if local parallel statements could **obtain** their remote resources in **parallel**. It was stated that such concurrency would not compromise the correctness of the system, since only one parallel statement would **execute** at a time due to the fact that there was only one processor for the set of local parallel statements.

When multiple processes may run on a processor, the efficiency is improved even further, because **all** parallel statements that are **located at different processors** and that **do not share objects** could execute simultaneously. Recall that in a single process per processor distributed architecture, parallel statements can execute simultaneously if they are **located at different processors**, they **do not share objects** and the **target objects** referenced by the respective statements **do not share processors**. Since the statement which handles messages from remote objects can be assigned to a separate process if each processor has multiple processes running on it, the last condition is removed.

Reuse

It is evident from the above that the infrastructure required to map a SLOOP program to a distributed architecture is not trivial. However, once such an infrastructure has been developed, it can be reused whenever a SLOOP program needs to be mapped to such an architecture.

8.3.4.2 Identifying the objects that need to be reserved

The next issue to consider is **how to determine which objects should be reserved** for each parallel statement. This is done by inspecting the parameters used in the message expressions in the parallel statements. The receiver of the parallel statement, as well as all target objects specified as parameters have to be reserved.

If a statement only sends messages to local objects, no objects need to be reserved if all the objects can be allocated to the statement simultaneously. If a statement executes under such circumstances, it cannot interfere with any other statement, as illustrated by the examples in the previous section.

As discussed in Chapter 4, Section 4.3.5.4, parallel statements may be nested. The purpose of the parallel statements at the top nesting level at each processor is to invoke the required parallel methods of the local objects. These methods may send messages to other objects. In the SLOOP method it is a requirement that **such target objects** have to be **named explicitly as parameters**

of these methods if they do not form part of the receiver. Thus, a composite object may refer to its constituent objects without having to name them as parameters in its methods. However, any other target object has to be passed to the sending object as a parameter. When a composite object is reserved, all its components are reserved with it.

Reservation requests are only issued at the processor where the top nesting level of a parallel statement is located. Since all target objects that do not form part of the receiver of the parallel statement have to be passed as parameters to the receiver, the target objects that are involved in the execution of a parallel statement can easily be determined when the statement is selected for execution.

Due to the fact that SLOOP parallel statements may be nested, and each parallel statement may result in multiple parallel statements at the next nesting level, it is possible that all the parameters that are passed by the top level parallel statement might not be required by each parallel statement at the bottom level. This is exemplified by the parallel methods of the TimerServices[18] class.

At the top nesting level the `p_runTimer:` method is invoked with `timerEventQ` as parameter, as shown below.

```
timer p_runTimer: timerEventQ
```

The p_runTimer: method contains three parallel statements, viz.

```
    currentTime := SmalltalkLibPkg:::Time now asSeconds
[] lastTime := currentTime \+
   currentTick := (currentTick + 1) \\ (timeoutCollection size)
       if difference ≥ 1 and: [currentTimeoutElement isNil]
[] timerEventQ addLast: currentTimeoutElement \+
   currentTimeoutElement updateEndTime \+
   currentTimeoutElement timerServicesCompleted: true \+
   (timeoutCollection at: readIndex) removeFirst
       if currentTimeoutElement notNil
```

As is evident from the above, only the third statement refers to `timerEventQ`. In order to improve efficiency, the reservation requests are not made until the parallel method at the top level has been expanded to its lowest level and the relevant statement for that particular pass has been selected. This expansion procedure forms part of the mapping of a SLOOP program to a distributed architecture.

If, in the above example, the `timerEventQ` object is located at a remote processor, then no remote objects need to be reserved when the first two statements are selected for execution. These statements may be executed whenever they are selected if the `TimerServices` instance is not allocated to a remote parallel statement at the time of their selection.

The third statement requires both the `TimerServices` instance and the `timerEventQ` object to be allocated to it before it may execute. This will involve a reservation request to another processor if the `timerEventQ` object is located remotely.

Parallel messages to the pseudo-variable `self` are used for structuring purposes. It is used to invoke parallel methods that contain statements that are likely to change during subclassing. A parallel message to the variable `self` is not required to pass its own instance variables to itself as arguments. If such a message is encountered at the top nesting level it is necessary to expand the

[18] The SLOOP specification of the TimerServices class is given in Appendix B, Section B.11.

parallel statement containing this message in order to determine which objects should be reserved for the resulting statements.

For example, the `p_activate` method of the `CC_Activation`[19] class contains the following parallel statement:

```
self p_executeCPAgent
```

The `p_executeCPAgent` method of the `CC_SimulationActivation`[20] subclass contains the following parallel statements:

```
commsAgent p_simulate: timer timeoutEventsIn: timerEventQ
[] commsAgent p_generateEvent: userConnections target: inputQ
```

Thus, it is evident that the `commsAgent`, `timer` and `timerEventQ` objects are involved in the first statement, while the `commsAgent`, `userConnections` and `inputQ` objects are involved in the second statement. Since these are all instance variables of the `CC_Activation` class, it does not have to pass these parameters to itself when it sends a message to itself. It is only by expanding the statement containing the pseudo-variable `self` that it is possible to determine which objects are required for the execution of that statement .

8.3.4.3 The middleware infrastructure

Due to the advent of middleware products such as CORBA [OHE97], there is no need to be concerned about issues such as how to determine the location of an object when a message has to be sent to it. When a statement in a distributed object implementation is selected for execution, the CORBA services are used to determine the location of all the target objects involved. Although the CORBA Concurrency Control Service [OHE97] allows the designer to lock resources, this service is not used by SLOOP, since it does not provide the functionality required for the reservation of objects as described above. For example, when a lock is requested via the CORBA Concurrency Control Service, the requesting process blocks until the lock is granted, whereas the SLOOP implementation requires that it should be possible to issue reservation requests asynchronously.

Once a parallel statement starts executing, the Object Request Broker (ORB) intercepts all messages sent to a class or an instance. It takes care of locating the target object and of converting the message selector and its argument to a format that can be transmitted to a remote processor.

There are various ways of incorporating CORBA into an implementation: one option suggested in [OHE97] is to multiply inherit from the CORBA services classes. If multiple inheritance is not supported by the target architecture, CORBA allows the inclusion of *before* and *after* callbacks that are executed before and after each method respectively [OHE97]. The necessary CORBA services can be invoked from within these callbacks. The latter approach is followed in the SLOOP method, since multiple inheritance is not supported.

This concludes the discussion of the **basic** principles involved during the mapping of SLOOP programs to different types of architectures. The remainder of this chapter deals with various topics related to such mappings. For example, it is shown how the concept of reflective computation can be utilised in SLOOP mappings. There is also a discussion on how more parallelism can be introduced into a SLOOP design if that is found to be a requirement during the implementation phase. However, first more detail is given regarding the mapping of macros and different types of SLOOP statements.

---

[19] The CC_Activation class is defined in Appendix B, Section B.2.
[20] The SLOOP specification of the CC_SimulationActivation class is presented in Appendix B, Section B.3.

8.4 Mapping macros

The previous sections described how the SLOOP **computational model** can be mapped onto various architectures. However, there are other SLOOP constructs that also need consideration during the mapping process. This section covers the issues regarding the mapping of the *macros-section* of a SLOOP class or method to Smalltalk.

In Chapter 4, Section 4.4.3.3, the simplest mapping approach was presented. Each *macro-variable* is simply replaced with its corresponding *macro-expression* wherever it is used. In order to improve efficiency, other options can be considered. However, some of these options are only more efficient under very specific circumstances and could even be less efficient in others, as shown below.

The first option is as follows: For each method all the *macro-variable*s are declared as temporary Smalltalk variables. This includes the *macro-variable*s defined in the class *macros-section* that are referenced by the specified method, as well as those in the *macros-section* of the specified method. If a *macro-expression* contains another *macro-variable*, the latter has to be declared as a temporary variable as well. The *macro-expression*s are evaluated and the results are assigned to the corresponding *macro-variable*s when a parallel or sequential method is **entered**.

This approach is more efficient than the mapping given in Chapter 4, Section 4.4.3.3, if the same *macro-variable* is referenced multiple times in a sequential method or if it is referenced multiple times within the same statement in a parallel method. However, this mapping only produces the correct results if the *macro-expression* has the same value at all locations in the method. In the case of a parallel method, where only one statement is selected for execution at each invocation, the *macro-variable*s may not even be referenced in the selected statement. In that case the evaluation of the *macro-expression* is a wasted computation.

Another option is to use a **hybrid** approach, i.e. in **sequential** methods macros are evaluated **once** when the method is **entered**, while in **parallel** methods the *macro-variable*s are replaced with their corresponding *macro-expression*s **wherever they are used**. Again this solution can only be used if each *macro-expression* always has the same value regardless of the location in the sequential method. Thus, if the *macro-variable* has a different value at different occurrences within a sequential method, a temporary variable mapping will not preserve the semantics of the SLOOP statements.

The simplification of the implementation of correctness property checks is another argument in favour of merely replacing *macro-variable*s with their corresponding *macro-expression*s wherever they are used. If the *properties-section* is implemented using reflection (as discussed in Section 8.6.3), then the *macro-expression*s in the property specifications are evaluated from within the metaclass by obtaining the relevant values of class and instance variables from the base class. The alternative options above map the *macro-variable*s to temporary variables. The scope of the temporary variables is the method within which they appear, i.e. at the time when the **preconditions** are checked in the metaclass no values have been assigned to these temporary variables yet. However, since it is **optional** to implement the *properties-section*s of a SLOOP program, this will not be a consideration if the *properties-section*s are not mapped to the executable program.

Thus, the simplest mapping of *macro-variable*s would be to replace them with their *macro-expression*s wherever they are used. In order to improve the efficiency of the algorithm, temporary variables could be used in **some** situations, but the designer would have to take care that the **semantics** of the SLOOP statement are preserved.

8.5 Mapping SLOOP statements

The mapping of a SLOOP statement which contains a single *statement-component* and which, in turn, contains a single *component-part* is straightforward as was demonstrated in Chapter 4, Section 4.4.3.3. This section deals with the Smalltalk mapping of parallel methods that contain multiple *quantified-statement-lists*. It also covers the mapping of statements comprising multiple *statement-components* and *component-parts*.

8.5.1 Mapping quantified-statement-lists

When multiple *quantified-statement-lists* appear in a **parallel** method, each statement within each quantification has to be included in the list of statements that are executed infinitely often. It has to be included in such a way that only one statement is selected at each invocation of the method. In the code fragment below, two of the *quantified-statement-lists* appearing in the `p_activate` method of the `CC_Activation class`[21] are shown.

```
[] < [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
        >
[] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
      p_execute
        >
```

When the class is instantiated, the `p_activateTally` instance variable is set to the total number of parallel statements contained within the `p_activate` method. For brevity some of the statements in the `p_activate` method are omitted in this example. The mapping below only shows the statements above, as well as two other enumerated statements.

The `p_activateTally` variable is set to `2 + (config maximumServiceCategories) + (config maximumConnections)` in the `initialize` method. In the `p_activate` method, the first statement calculates which parallel statement should be executed. The second statement calculates which of the quantified statements should be executed, should the current value of `p_activateCycleIndex` not indicate one of the enumerated statements. The temporary variables `i` and `j` are used to select the correct instances of the `Connection` and `ServiceCategory` classes respectively.

```
"Determine which statement should be executed."
p_activateCycleIndex :=
      ((p_activateCycleIndex + 1) \\ p_activateTally).

"Determine whether it is one of the statements of the
Connection class"
(p_activateCycleIndex > 1 and:
[p_activateCycleIndex ≤ (1 + config maximumConnections)])
      ifTrue: [i := i+1]
      ifFalse:
      [
            i := 0.
            "Determine whether it is one of the statements of
            the ServiceCategory class"
            (p_activateCycleIndex >
            (1 + config maximumConnections) and:
            [p_activateCycleIndex ≤
```

[21] The SLOOP specification of the CC_Activation class is presented in Appendix B, Section B.2.

```
                (1 + config maximumConnections +
                config maximumServiceCategories)])
                      ifTrue:  [j := j+1]
                      ifFalse: [j := 0].

(p_activateCycleIndex = 0)
ifTrue: [self p_executeCPAgent]                    "statement 0"

ifFalse: [(p_activateCycleIndex = 1)
      ifTrue: [timer p_runTimer: timerEventQ]      "statement 1"

      ifFalse: [ (p_activateCycleIndex = (1+i))
            ifTrue: [self p_executeConnection:
                  (userConnections at: i)]      "statement 1+i"

            ifFalse: [ (p_activateCycleIndex = (1 +
                        config maximumConnections + j)
                  ifTrue: [(scContainer at: j) p_execute ]
                  "statement 1 + config maximumConnections + j"
                  ifFalse:[...
                        ]
                  ]
            ]
      ]
```

The above code fragment represents one possible way of mapping *quantified-statement-lists*. Many other mappings are possible, but the above suffices as an illustration of how it could be done.

8.5.2 Mapping statement-components and component-parts

The next example illustrates the mapping of a SLOOP statement comprising multiple *statement-components* and *component-parts*. This statement from the `p_generateEvent:target:` method of the `CommsProviderSimulator` class[22] has two *statement-components* separated by the '||' symbol. (For the purposes of this example, the two *statement-components* appear in a specific order in this section, which happens to be different from their ordering in Appendix B. This is acceptable, since the order in which *statement-components* appear in a SLOOP program is not significant.)

```
newEventRequired := true \+
generatingEvent := false
      if generatingEvent
||inputQ addLast: (idleConnection serviceRequest) \+
idleConnection assign
      if generatingEvent and: [idleConnection notNil] ~
Transcript show: 'All connections busy'
      if generatingEvent and: [idleConnection isNil]
```

(The *macro-variable* `idleConnection` is defined as `self getIdleConnection: userConnections`.)

It is important that the semantics of the statement should be preserved, otherwise the resulting program could be incorrect. The evaluation order of a SLOOP statement was first presented in Chapter 4, Section 4.3.6.3. Recall that all conditional expressions appearing in a statement are evaluated first. Thereafter the message expressions that represent receivers and arguments are

[22] The CommsProviderSimulator class is specified in Appendix B, Section B.6.

evaluated. Finally, the resulting message expressions (the only ones that may modify variables) are evaluated.

If, instead of following these rules, the statement above is mapped to a sequential architecture by executing the *statement-component*s sequentially, `generatingEvent` would be set to false before the conditional expressions of the second *statement-component* are evaluated. No service requests would ever be added to the `inputQ`, even if there are idle connections, as can be seen below.

```
(generatingEvent)
ifTrue:
[
      newEventRequired := true.
      generatingEvent := false
].
(generatingEvent and:[(self getIdleConnection: userConnections)
 notNil])
ifTrue:
[
      inputQ addLast:
      (self getIdleConnection: userConnections) serviceRequest.
      (self getIdleConnection: userConnections) assign
]
ifFalse: [Transcript show:'All connections busy'].
```

A correct mapping of the above statement for a sequential architecture is the following:

```
(generatingEvent)
ifTrue:
[
      newEventRequired := true.
      generatingEvent := false.
      ((self getIdleConnection: userConnections) notNil)
      ifTrue:
      [
            inputQ addLast:
            (self getIdleConnection: userConnections) serviceRequest.
            (self getIdleConnection: userConnections) assign
      ]
      ifFalse: [Transcript show:'All connections busy'].
]
```

There are several correct mappings for the above statement. The important issue is to make sure that if the value of a variable is **used** and **modified** in the **same** statement, then its value **prior** to the modification has to be obtained for all occurrences where it is **used**. All of these occurrences have to use the value obtained **prior** to the modification.

A modification may be performed explicitly via an assignment that forms part of the parallel statement, or it may occur implicitly via an assignment that is performed as a result of a method being invoked from within the parallel statement. When `generatingEvent` is set to false in the above statement, it is an example of an explicit modification. The invocation of the `assign` method of the `Connection` class in the above statement is an example of an implicit modification.

Note that in a synchronous shared-memory architecture as described in Sections 8.2.2 and 8.3.2, each *component-part* of a parallel statement can be assigned to a different processor. Thus, the `addLast:` and the `assign` methods in the above statement can be executed simultaneously, and at the same time the values of `newEventRequired` and `generatingEvent` can also be

updated. The various *statement-component*s and *component-part*s of a parallel statement may therefore be executed concurrently, with the proviso that **all** read accesses performed by any *statement-component* or *component-part* of the statement are completed before any write access may commence.

8.6 The use of reflection in mappings of SLOOP programs

In Chapter 1, Section 1.3.4, it was stated that computational reflection could be used, inter alia, to **reason about control** and for **assertion checking**. These topics are now explored further.

Each time when a parallel method is invoked, only one of its constituent statements is executed. In the mapping to an executable Smalltalk program additional variables and logic are introduced in order to select a statement for each invocation. In Section 8.6.2 it is shown how this aspect of the mapping can be delegated to the metaclass of each base class, thereby ensuring that the latter is not cluttered with variables and statements that are irrelevant to the class itself.

When a method from the base class is invoked, it is possible to check the preconditions for that method in the metaclass before the method is executed. Once the method has completed, the postconditions can be checked in the metaclass before control is returned to the client. This aspect of the use of reflection in the SLOOP method is covered in Section 8.6.3. However, first more information is given regarding the infrastructure that is required in order to make use of reflection in this way.

8.6.1 A reflective computation infrastructure

The ALBEDO meta-object infrastructure [Bekk93] is one possible infrastructure that can be used to provide reflective computational facilities. The basic principle employed in a meta-object infrastructure is to associate a metaclass with a base class and to allow the metaclass object to **intercept** the messages sent to the base class object.

The ALBEDO meta-object infrastructure provides a **mechanism** to intercept these messages. It is implemented in Smalltalk-80. The Smalltalk architecture consists of a virtual machine and a virtual image [GoRo89]. The virtual machine handles the interface to the hardware and also contains low level routines that must be written in machine language. The virtual image consists of the kernel objects, the compiler objects and the users' objects. Since the **message handling** primitive of Smalltalk is part of the virtual machine, it is inaccessible to users.

The way in which this problem is overcome in the ALBEDO system is by **encapsulating** the base object and its associated meta-object in a **shell**. A minimal set of methods is implemented for the class representing this shell. As a result most messages received by the shell are not understood. The `doesNotUnderstand:` method is reimplemented in the shell. Instead of displaying an error message, the `handleMsg: aMessage` message is sent to the encapsulated meta-object, where `aMessage` is the original message received by the shell.

The meta-object takes the necessary actions pertaining to that particular metaclass and it then passes the original message to its associated base object. Depending on the function of the meta-object, it may even modify the original message before sending it to the base object. This aspect will be discussed in more detail below. A graphical representation of the architecture of the infrastructure is presented in Figure 8-10.

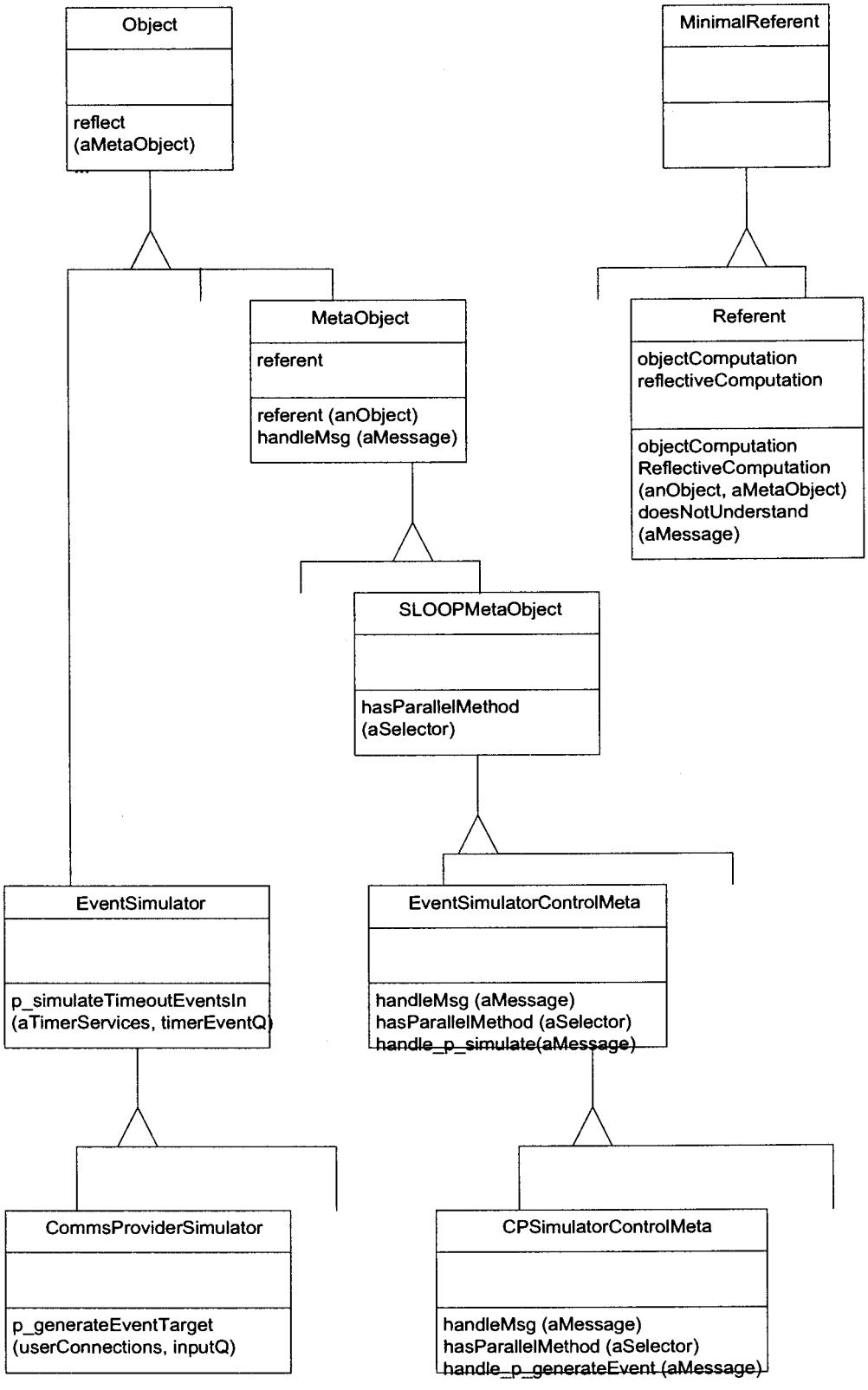

Figure 8-10. Class diagram of the ALBEDO meta-object infrastructure based on Smalltalk.

Only the most important instance variables and methods relevant to the discussion are shown in the diagram. The `EventSimulator` and `CommsProviderSimulator` classes[23] are used as examples of a base class hierarchy. The `Object` class is their root class. To a large extent the metaclass hierarchy mimics the base class hierarchy. However, all metaclasses have to be descendants of the `MetaObject` class. The latter is a subclass of `Object`. It is also not mandatory for each base class to have a metaclass associated with it. The purpose of the `SLOOPMetaObject` class is to add methods and instance variables that are specific to SLOOP mappings. This class and its descendants are described in more detail later on in this section.

The **implementation of the shell** is realized by two new classes, viz. `MinimalReferent` and `Referent`. `MinimalReferent` is a new root class. A minimal set of methods from the Smalltalk `Object` root class is recompiled for `MinimalReferent`. The subclass `Referent` contains the functionality specific to the implementation of reflective computation. This class contains the two instance variables `objectComputation` and `reflectiveComputation`. Once the encapsulation has been performed via the new `reflect:` method of the `Object` class, they refer to the base object and meta-object respectively. The `reflect:` method will be discussed in more detail shortly. The `Referent` class also contains the reimplemented `doesNotUnderstand:` method. Reflective computation is activated as illustrated by the example of the `CommsProviderSimulator` class given next.

The `initialize` method of the `CC_Activation` class contains the following statement:

```
commsAgent := self initCommsAgent
```

The `initCommsAgent` method is defined as being the responsibility of the subclass (in this case `CC_SimulationActivation`). In the original `CC_SimulationActivation` class this method contains the following statement:

```
^CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation
```

The `initCommsAgent` method is now augmented with the statements related to reflective computation (the additions are shown in bold):

```
|commsAgentBase commsAgentMeta|
commsAgentBase :=
      CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation.
commsAgentMeta :=
      CC_SimulationInterfacesPkg:::CPSimulatorControlMeta new.
commsAgentBase := commsAgentBase reflect: commsAgentMeta.
^commsAgentBase
```

The `reflect:` method is a **new method added to the `Object` class** by the ALBEDO meta-object infrastructure. Its purpose is to **encapsulate** the base object and its meta-object in a shell. The implementation of the `reflect:` method is as follows:

```
"Statement(s) of the reflect: method"
^Referent objectComputation: self reflectiveComputation: aMetaObject
```

Thus, the `objectComputation:reflectiveComputation:` class method of the `Referent` class is invoked. This method creates and returns a new instance of `Referent` and it causes the instance variables `objectComputation` and `reflectiveComputation` to refer to the encapsulated base object and meta-object respectively. At this stage the `referent` instance variable of `commsAgentMeta`, the meta-object, is also set to refer to `commsAgentBase`, its

[23] The EventSimulator and CommsProviderSimulator classes are specified in detail in Appendix B, Sections B.5 and B.6 respectively.

associated base object. (The `CPSimulatorControlMeta` class inherits the `referent` instance variable from the `MetaObject` class.) The base object has no reference to its encapsulation or its associated meta-object.

In this example the `commsAgentMeta` meta-object is an instance of the `CPSimulatorControlMeta` class. This class performs reflective computation related to the **control** of parallel statements in the base class. It is possible to instantiate a different type of metaclass at this point. For example, a metaclass that performs reflective computation related to assertion checking or one that performs both control actions and assertion checking could be instantiated. In the remainder of this section the general concepts are explained using the control metaclasses as examples.

In the program fragment above the `commsAgentBase` variable is set to the value returned by the `reflect:` method. This is also the value that is returned by the modified version of the `initCommsAgent` method. Thus, instead of referring to a `CommsProviderSimulator` instance, the `commsAgent` instance variable refers to the **shell** encapsulating the `CommsProviderSimulator` instance and its associated meta-object. (The shell is an instance of `Referent`.)

When a message is now sent to `commsAgent`, the **shell** receives it. Since the shell understands only a minimal set of messages, the `doesNotUnderstand:` method is invoked for most messages. The `doesNotUnderstand:` implementation for the `Referent` class is as follows:

```
^reflectiveComputation handleMsg: aMessage.
```

Thus, the message is passed to `commsAgentMeta`, the meta-object. The `handleMsg:` method now performs the reflective computation before it passes the message to the base object.

(The `selector` and `respondsTo` : methods invoked in the program fragment below refer to Smalltalk-80 library methods and are not shown in Figure 8-10. The receiver of the `selector` method is a message expression. The `selector` method returns the selector found in this message expression. For example, the `(inputQ addLast: serviceRequest) selector` message expression would return the `addLast:` selector. Thus, if `(inputQ addLast: serviceRequest)` is passed as argument to the `handleMsg:` method below, then `aMessage` would have the value `(inputQ addLast: serviceRequest)` and `msgSelector` would have the value `addLast:`. The `respondsTo:` method returns true if the receiver supports the method specified as argument of the `respondsTo:` method.)

The `handleMsg:` method of `CPSimulatorControlMeta` is as follows:

```
|msgSelector|
msgSelector := aMessage selector.
(referent respondsTo: msgSelector)
ifTrue:
[
        (msgSelector = #p_generateEvent:target:)
        ifTrue:
                [^self handle_p_generateEvent: aMessage]
        ifFalse:
        [
                (super hasParallelMethod: msgSelector)
                ifTrue: [^super handleMsg: aMessage]
                ifFalse:
                [^referent perform: msgSelector
                withArguments: (aMessage arguments)]
        ]
]
```

```
ifFalse:
        [^referent doesNotUnderstand: aMessage]
```

First of all, the meta-object checks whether the base-object can respond to the specified message selector. If it cannot, the `doesNotUnderstand:` method associated with the base-object is invoked (i.e. the original one implemented in the `Object` class, which displays an error message).

If the message can be understood, the meta-object performs its reflective computation. In the case of a `CPSimulatorControlMeta` instance, which needs to take control actions, the meta-object then checks whether it indicates one of the parallel methods of the base class (in this case there is only one, viz. `p_generateEvent:target:`).

If it does, `handle_p_generateEvent:`, the meta-object method which handles the selected parallel method, is invoked. The contents of `handle_p_generateEvent:` is discussed in Section 8.6.2. At this stage it suffices to note that the method selects one of the parallel statements of the `p_generateEvent:target:` method and ensures that the selected statement is executed.

If the message selector does not match one of those supported by the base-object, the superclass of the current meta-object is consulted. If the superclass (or one of its ancestors) can find a match for the specified message selector, the message is passed to the relevant ancestor. If no match can be found, it is assumed that the message selector represents a sequential method and the message is passed to the base-object using explicit message passing.

The `CommsProviderSimulator` class inherits the `p_simulate:timeoutEventsIn:` method from `EventSimulator`, its parent class. The reflective computation related to this method is found in the corresponding metaclass. Details regarding the `handle_p_simulate:` method will be given in Section 8.6.2.

The `handleMsg:` method of `EventSimulatorControlMeta` is as follows:

```
|msgSelector|
msgSelector := aMessage selector.
(referent respondsTo: msgSelector)
ifTrue:
[
        (msgSelector = #p_simulate:timeoutEventsIn:)
        ifTrue:
                [^self handle_p_simulate: aMessage]
        ifFalse:
        [
                (super hasParallelMethod: msgSelector)
                ifTrue:
                        [^super handleMsg: aMessage]
                ifFalse:
                [^referent perform: msgSelector
                withArguments: (aMessage arguments)]
        ]
]
ifFalse:
        [^referent doesNotUnderstand: aMessage]
```

The class `MetaObject` is the superclass of all meta-classes. The subclass `SLOOPMetaObject` merely adds the method which checks whether the associated base class contains any parallel methods. In `SLOOPMetaObject` the method is implemented as follows:

```
hasParallelMethod: aSelector
^false
```

Each subclass of `SLOOPMetaObject` which performs reflective computation about the parallel statements in its associated base class has to reimplement this method. Thus, in `EventSimulatorControlMeta` (subclass of `SLOOPMetaObject`) it is reimplemented as shown below:

```
hasParallelMethod: aSelector
(aSelector = #p_simulate:timeoutEventsIn:)
ifTrue: [^true]
ifFalse: [^super hasParallelMethod: aSelector]
```

`CPSimulatorControlMeta` (subclass of `EventSimulatorControlMeta`) also reimplements the method:

```
hasParallelMethod: aSelector
(aSelector = #p_generateEvent:target:)
ifTrue: [^true]
ifFalse: [^super hasParallelMethod]
```

In the next section it will be shown how reflective computation is used to **select a parallel statement** for execution.

8.6.2 Using reflective computation for control purposes

In Section 8.3.1 it was described how the selection of a single parallel statement per parallel method invocation can be achieved on a sequential architecture. The mechanism is based on the introduction of two additional variables. The one contains the total number of statements within the method and the other variable is used to record which statement had been executed during the previous cycle.

Instead of adding these variables to the base classes, they are added to the meta-classes. Each meta-object has to initialize these variables. If such variables are inherited from a superclass, it has to be ensured that the variables in the superclass are initialized as well. This is illustrated by the example below. The `new` method of the `EventSimulatorControlMeta` class is implemented as follows:

```
^(super new) eventSimulatorInit
```

The `eventSimulatorInit` method initializes the `p_simulateTally` and `p_simulateCycleIndex` variables:

```
"Statements of the eventSimulatorInit method"
p_simulateTally := 2.
p_simulateCycleIndex := p_simulateTally - 1.
```

The `CPSimulatorControlMeta` metaclass is a subclass of `EventSimulatorControlMeta`. Thus, when this class is instantiated, it has to ensure that the instance variables inherited from its superclass are initialized as well. Its new method is therefore implemented as follows:

```
^(super new) cpSimulatorInit
```

Thus, first of all the `new` method of `EventSimulatorControlMeta`, its superclass, is invoked. That results in an instance being created, as well as in the invocation of the

`eventSimulatorInit` method. The instance variables of its superclass are therefore initialized. Subsequently the `cpSimulatorInit` method is invoked. The latter initializes the instance variables of the `CPSimulatorControlMeta` class, as shown below:

```
"Statements of the cpSimulatorInit method"
p_generateEventTally := 1.
p_generateEventCycleIndex := p_generateEventTally - 1.
```

When a message is received by the shell and it has been passed to the encapsulated meta-object via the `handleMsg:` method, the meta-object determines whether the message selector indicates a parallel method or a sequential method. It also determines whether it is a parallel method supported at the current level in the class hierarchy or by an ancestor. If it is a parallel method selector, the relevant meta-object method is then invoked to perform the selection of the appropriate statement. This procedure was covered in the previous section. The implementation of the meta-object methods that perform the statement selection is now described.

The purpose of moving the parallel statement selection (i.e. **control**) functionality to the metaclass level is to keep the mapping of the base class as close as possible to the original SLOOP statements. However, there is one aspect that cannot be kept transparent to the base class, viz. the fact that it has to be possible to refer to a single parallel statement at a time.

If a parallel method contains multiple statements (which may be convenient for abstraction purposes), the Smalltalk mapping has to include a separate method for each parallel statement within the method. These methods are not visible to any other base classes. Thus, all other base classes refer to the parallel method that corresponds with the SLOOP parallel method. The methods containing the single parallel statements are only used by the corresponding metaclass. This is now exemplified by the `EventSimulator` and `CommsProviderSimulator` classes.

The SLOOP version of the statements in the `p_simulate:timeoutEventsIn:` method is shown first. It contains two parallel statements:

```
self startRandomTimer: aTimerServices withMaximum:
(aTimerServices maximumTimeout) \+
newEventRequired := false
      if newEventRequired
[] generatingEvent := true \+
self resetTimerExpired: timerEventQ
      if self timerExpired: timerEventQ
```

The `handle_p_simulate:` aMessage method of the `EventSimulatorControlMeta` class is implemented as follows:

```
handle_p_simulate: aMessage

p_simulateCycleIndex := (p_simulateCycleIndex + 1) \\ p_simulateTally.
args := aMessage arguments.

(p_simulateCycleIndex = 0)
if True:
[
      ^referent perform: (#p_s1_simulate:) withArguments: (args at: 1)
]
ifFalse:
[
      (p_simulateCycleIndex = 1)
      ifTrue:
            [^referent perform: (#p_s2_simulate:)
```

```
            withArguments: (args at: 2)]
]
```

The corresponding single statement methods in the base class are as follows:

```
p_s1_simulate: aTimerServices

(newEventRequired) ifTrue:
[
      self startRandomTimer: aTimerServices withMaximum:
      (aTimerServices maximumTimeout).
      newEventRequired := false
]


p_s2_simulate: timerEventQ

(self timerExpired: timerEventQ) ifTrue:
[
      generatingEvent := true.
      self resetTimerExpired: timerEventQ
]
```

Note that the original `p_simulate:timeoutEventsIn:` method contained two arguments. However, each of its constituent parallel statements only refers to one of these arguments. This is reflected by the new methods invoked by the metaclass.

The `handle_p_generateEvent:` aMessage method of the `CPSimulatorControlMeta` class is implemented as follows:

```
handle_p_generateEvent: aMessage

p_generateEventCycleIndex :=
      (p_generateEventCycleIndex + 1) \\ p_generateEventTally.
args := aMessage arguments.

(p_generateEventCycleIndex = 0)
if True:
      [^referent perform: (aMessage selector): withArguments: args]
```

Note that in this case the parallel method in the SLOOP class contains only one statement as shown below, in which case there is no need to define additional single statement methods.

```
      inputQ addLast: (idleConnection serviceRequest) \+
      idleConnection assign
            if generatingEvent and: [idleConnection notNil] ~
      Transcript show: 'All connections busy'
            if generatingEvent and: [idleConnection isNil]
      || newEventRequired := true \+
      generatingEvent := false
            if generatingEvent
```

The ALBEDO meta-object infrastructure does have the limitation that messages to the Smalltalk pseudo-variables `self` and `super` are not intercepted [Bekk93]. The reason for sending a parallel message to `self` is purely for structuring purposes. Equivalent functionality is achieved by replacing the statement containing the message to `self` with the constituent statements of the corresponding method. Thus far there has been no requirement for sending parallel messages to

`super`. An example of parallel statements containing messages to `self` can be found in the `p_activate` method of the `CC_Activation` class in Appendix B, Section B.2. This concludes the discussion on how reflective computation can be used to control which parallel statement should be executed next. In the next section another application of reflective computation is covered, viz. assertion checking.

8.6.3 Using reflective computation for assertion checking

It is of the utmost importance to check that the relevant correctness properties are not violated by the mapping of any SLOOP program fragment to an executable program. Some of these properties can be checked at run-time using the reflective facilities of Smalltalk. In particular, each message can be intercepted by the meta-object of the target object. The preconditions are checked before the message is passed to the target object. When the latter has completed its execution of the corresponding method, control returns to the meta-object, which then checks the postconditions before returning control to the client object. The ALBEDO meta-object infrastructure [Bekk93] as described in Section 8.6.1 can again be used to achieve this.

However, the **liveness** and **precedence** properties of a parallel method cannot be checked in this way. The postconditions of these properties are only required to hold eventually, provided the preconditions hold at some point and the method is invoked infinitely often. During the implementation phase it is therefore necessary to ensure that each parallel statement will indeed be executed infinitely often. In Section 8.2 it was described how this could be guaranteed by enclosing the relevant statements in infinite loops. It is imperative to check that the variables representing the total number of statements in each parallel method contain the correct values. The algorithm used to select the next statement within a parallel method has to guarantee that each statement within the method will eventually be selected. The software designer therefore has to take special care that these aspects of the correctness of the mapping are thoroughly checked.

Apart from reasoning about the correctness of the mapping, it is also possible to implement trace statements as part of the meta-objects. In so doing, the statements of the base objects are not affected, while an additional mechanism is provided to increase confidence in the correctness of the mapping.

Computational reflection is therefore a powerful mechanism that can be used during the implementation phase of a project based on the SLOOP method. It facilitates the separation of concerns, i.e. the base classes represent the SLOOP classes, while the metaclasses contain the information about those classes that are implicitly present in the SLOOP classes.

8.7 Modifying the level of parallelism in a SLOOP design

In some cases, depending on the target architecture to which the SLOOP program should be mapped, it could be found during the implementation phase that more parallelism in the design would be beneficial. The SLOOP approach makes it relatively easy to introduce more parallelism into a design.

For example, when the `ServiceCategoryAllocator`[24] instance needs to categorise the service request at the head of the `inputQ`, it does this by executing the following parallel statement:

```
categorising := true \+
self categoriseServiceRequest: (inputQ first) using: scContainer
      if inputQ isEmpty not and: [categorising not]
```

[24] The ServiceCategoryAllocator class is specified in Appendix B, Section B.8.

The purpose of the `categorising` instance variable is to ensure that the `categoriseServiceRequest:using:` method is only invoked once for the service request at the head of the `inputQ`. Once the service request has been categorised, another parallel statement[25] is executed which removes it from the `inputQ` and at the same time sets the `categorising` variable to false, thereby facilitating the `categorisation` of the next element in the `inputQ`. Depending on the requirements of the system, the categorisation of a service request could be lengthy and complex.

One disadvantage of the formulation of the above statement is the fact that in order to categorise an entry from the `inputQ`, all the objects that are involved at various stages of the categorisation have to be reserved. Thus, `scAllocator`, `inputQ` and all its elements, as well as `scContainer` and all its elements have to reserved before this statement can be executed.

It is desirable to reduce the number of objects that have to be reserved for a particular statement, since that would decrease the period for which those objects are tied up. This is because they are not released until all of the relevant objects have been reserved and the statement has completed its execution.

A higher degree of parallelism is achieved by splitting the statement shown above into two statements, simply by introducing an additional instance variable, viz. `currentServiceRequest`. The resulting statements are shown next:

```
        categorising := true \+
        currentServiceRequest := inputQ first
            if inputQ isEmpty not and: [categorising not]
    [] self categoriseServiceRequest: currentServiceRequest
       using: scContainer
            if currentServiceRequest notNil
```

The `currentServiceRequest` instance variable is set to nil inside the `categoriseServiceRequest:using:` method.

The above design facilitates a higher level of parallelism. For example, the `CommsProviderSimulator` class statement which adds new service requests to the `inputQ` and the `ServiceCategoryAllocator` statement which invokes the `categorise=ServiceRequest:using:` method can be executed concurrently (provided these two statements or the objects that they refer to do not share processors). It is therefore quite clear that it is relatively simple to introduce more parallelism into a design. The only issue that has to be taken into account whenever such modifications are made, is the fact that none of the correctness properties should be violated by the modifications.

8.8 Summary

This chapter covered various aspects related to the implementation phase of the SLOOP method.

In earlier chapters it was often stated that the SLOOP method encouraged a **unified** approach towards system design; there was no need to consider the target architecture during the analysis and design phases. This chapter served to reaffirm this view. It demonstrated that a **single** SLOOP program could be mapped successfully to **sequential**, **synchronous shared-memory**, **asynchronous shared-memory** and **distributed** architectures.

[25] This second parallel statement is not shown here, but can be found in the p_allocate:from: method of the ServiceCategoryAllocator class specified in Appendix B, Section B.8

An important aspect of the SLOOP method which emerged from the discussion of the mapping to distributed architectures is the **high level of abstraction** of the SLOOP program. The designer may rely on the **atomicity** of each parallel statement when issues such as exclusive access to objects and the execution of critical sections have to be considered. The **mapping** to the executable program has to ensure this atomicity. In Section 8.3.4.1 it became apparent how much of the **complexity** of the total system was **delegated** to the supporting infrastructure when it was shown briefly **how** the atomicity could be guaranteed in a distributed system environment. A **further level of delegation of functionality** is the use of a **middleware product** such as CORBA to take care of issues such as the converting of the message selector and its argument to a format that can be transmitted to a remote processor.

Another advantage of the **separation of concerns** is the fact that the supporting infrastructure only needs to be developed once. Thereafter it can be **reused** by any SLOOP program. This greatly simplifies correctness reasoning. The designer using the SLOOP method only needs to consider the SLOOP statements. The atomicity of these statements can be relied upon, since the behaviour of the supporting infrastructure can be assumed to be correct once it has been proved. The correctness properties of the supporting infrastructure are therefore being reused by the designers of the SLOOP programs.

Ideally, a SLOOP development environment should include the required supporting infrastructures for various architectures. Building such an environment is one of the subjects for future research.

Other aspects of the mapping of a SLOOP program to an executable Smalltalk program include the mapping of the *macros-sections* and the various types of SLOOP statements. These discussions were included to demonstrate that it was possible to perform these mappings with **relative ease**. A SLOOP translator could automate all of these tasks. The design and implementation of such a translator to a given target language (e.g. Smalltalk) is another subject for further study.

As a step towards making the final executable program look as much as possible like the original SLOOP program, it was shown that the **reflective facilities** of Smalltalk could be used to **select the next parallel statement** for execution. Again, this functionality could form part of the development environment. Similarly, reflective computation could also be used to perform **assertion checking**.

In addition to emphasizing the high level of abstraction of a SLOOP program, the relative ease with which more **parallelism** could be introduced into a SLOOP program was also pointed out.

Throughout this chapter the importance of ensuring the **adherence to the specified correctness properties throughout the development life cycle** was stressed. Some measures that may be taken to avoid the violation of these properties during the implementation phase were described.

This chapter concludes the description of the SLOOP method as it applies to the various phases of the software development life cycle. The next chapter deals with the incorporation of design patterns into a SLOOP design. The use of design patterns is not mandatory in a SLOOP design, but it can greatly enhance the reusability of the components of the system. Chapter 9 demonstrates the compatibility of the SLOOP approach with the concept of design patterns.

CHAPTER 9

INCORPORATING DESIGN PATTERNS INTO A SLOOP DESIGN

9.1 Introduction

As noted by Buschmann et al. [BMRSS96], one can classify patterns as being either **architectural patterns**, **design patterns** or **idioms**. Definitions of the various types of patterns were given in Chapter 3, Section 3.3.1.

The architectural patterns and design patterns listed in [BMRSS96] and [GHJV95] provide examples of many different types of design problems. The **purpose** of this chapter is to demonstrate that the **SLOOP approach** can be **applied successfully** to a **variety of design problems**. Several architectural and design patterns described in the above-mentioned references are therefore taken as examples and it is shown how they can be incorporated into a system that is designed using the SLOOP method.

Some of the patterns are already present in the original design of the call centre system as presented in Appendix B, while others can be added to improve the **reusability** and **extensibility** of the system. In both cases the SLOOP-specific issues are highlighted. These deal mainly with adherence to the SLOOP computational model, i.e. it has to be ensured that all the **parallel statements** of a particular application are **executed infinitely** often.

The purpose of incorporating architectural and design patterns into a SLOOP design is not to model the problem domain more accurately, but to yield a more reusable and flexible solution. The application of some of the design patterns results in reducing the amount of subclassing required when instantiating the system for a particular application.

It is beyond the scope of this chapter to provide detailed descriptions of the design patterns referenced here. For more information about these patterns the bibliographical references given in the various sections below should be consulted.

9.2 Architectural patterns

9.2.1 Pipes and filters

The design of the call centre system is reminiscent of the Pipes and Filters architectural pattern. The latter is described in [BMRSS96]. Briefly, data is generated by a data source, it is processed by successive filters and finally reaches a data sink. The filters are connected via pipes. Thus, the data is stored in a pipe by the previous filter or the source. The next filter in the pipeline then obtains the data from the pipe for further processing. This pattern is applicable when data is

processed in sequential steps. A filter processes its input incrementally, i.e. it does not read all its input before it processes the data and starts producing output. This promotes **parallelism** and a **low latency**.

In the call centre example shown in Figure 9-1, the CommsProviderSimulator[1] instance acts as the source of the data that needs to be processed (in this case a new service request is added to the `inputQ` (the first pipe in the pipeline). The ServiceCategoryAllocator[2] instance (the first filter) obtains the service request from the `inputQ` and initiates the categorisation of the service request. Once the service request has been categorised, it is assigned to the appropriate `serviceQ` (the next pipe). The next filter is the relevant ServiceCategory[3] instance, which assigns the service request to an idle ServiceProviderSimulator[4] instance. The latter acts as the data sink in this example.

One of the **benefits** of the Pipes and Filters pattern is that these **components** can be **added, deleted or rearranged** as required. In the call centre example the ServiceCategoryAllocator instance obtains its input from a FIFO queue of service requests. The ServiceCategoryAllocator instance receives the reference to its input pipe as a parameter when its `p_categorise:using:` method is invoked. It is therefore a trivial matter to construct an application where the ServiceCategoryAllocator obtains its input from a different queue. The only requirement is that the new queue should contain service requests of the same type.

It is also fairly simple to add another filter and a pipe between the ServiceCategoryAllocator and its existing input pipe. The new filter would process the service requests from the existing input pipe. Its output pipe would serve as the new input pipe of the ServiceCategoryAllocator.

Another possibility would be to construct a system where the ServiceCategoryAllocator was replaced by a completely different filter.

The SLOOP design approach is particularly suited to this architectural pattern, because it is based on a concept of a number of parallel statements that execute infinitely often. A parallel statement has an **effect** (e.g. it performs the filter function) if its ***if*-clause is true** (e.g. there is an element in the input queue). Parallel statements that do not share processors and that do not send messages to the same objects may execute **concurrently**. This facilitates the realization of the goals of increased parallelism and low latency.

[1] The CommsProviderSimulator class is specified in Appendix B, Section B.6.
[2] The ServiceCategoryAllocator class is specified in Appendix B, Section B.8.
[3] The ServiceCategory class is specified in Appendix B, Section B.10.
[4] The ServiceProviderSimulator class is specified in Appendix B, Section B.13.

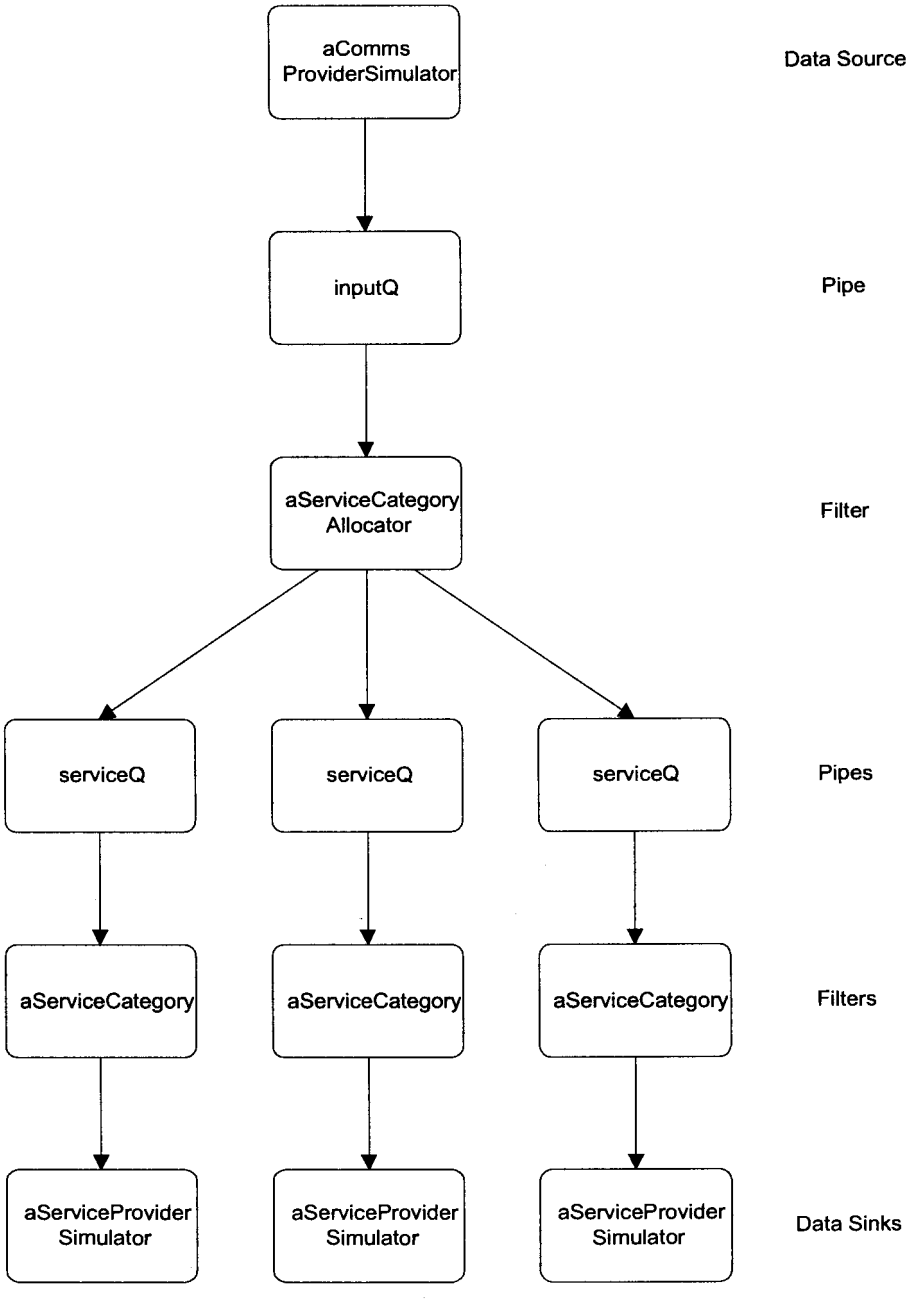

Figure 9-1. Pipes and filters in the call centre system.

9.2.2 Reflection

The reflection architectural pattern as described by [BMRSS96], provides a mechanism to change the behaviour of a software system by modifying the information at the meta-level. In CLOS, a reflective programming language [Keen89], the operations defined for an object are called generic functions. The invocation of such a generic function comprises a number of steps. First of all, the methods applicable to a given invocation of a generic function are determined, then they are sorted in decreasing order of precedence and subsequently a final sequence of methods is selected for execution [BMRSS96].

Similarly, in a SLOOP mapping which uses reflective facilities for its statement selection, the reflective computation **determines which of the private methods** associated with the parallel method **should be executed** at each parallel method invocation. (This was described in detail in Section 8.6.2 of the previous chapter.) A notable difference is that in a SLOOP mapping only one method may be selected for execution per parallel method invocation.

Some **assertion checking** can also be performed using reflective computation. Another possibility is the **generation of trace information**. These applications of reflective computation in the SLOOP method were discussed in Chapter 8, Section 8.6.3.

9.3 Creational design patterns

This section focuses on creational design patterns. The Factory Method and Singleton are two examples of creational design patterns [GHJV95]. The call centre example is used to demonstrate the **compatibility** between these patterns and the SLOOP approach towards system design.

9.3.1 The Factory Method

The CC_Activation class is used to instantiate the classes used by the call centre system and to ensure that it is done in the correct order. Evaluation of the design of the CC_Activation class as described in Chapter 6, Section 6.6.1, and in Appendix B, Section B.2, shows that it is already designed in the mould of two fundamental design patterns, viz. the Template Method (a behavioural design pattern) and the Factory Method (a creational design pattern). The aspects related to the Factory Method design pattern are described here. Section 9.5.3 covers the application of the Template Method design pattern.

The Factory Method design pattern is used when it is necessary for a class to **defer** the actual **instantiation** of certain classes to its subclasses. The **abstract class** has knowledge about the **sequence** in which it has to instantiate classes, but it needs to allow its **subclasses** to specify exactly **which** classes should be instantiated. Factory Methods are therefore used to instantiate those classes that are likely to be subclassed (or even replaced by other classes).

A Factory Method is usually invoked from within a **Template Method**. The latter indicates the **sequence** in which the classes should be instantiated. The Factory Methods are invoked at the various locations where these classes need to be instantiated. The **Factory Methods** therefore provide the **hooks for the instantiation** of these classes without committing to specific classes.

The `initialize` method of the CC_Activation class represents a Template method. It invokes several Factory Methods as can be seen below. Examples of objects that are created via Factory Methods are the `config` and `commsAgent` objects. In contrast the `userConnections` object is created directly by the CC_Activation class.

The statements of the `initialize` method of the CC_Activation class are as follows:

```
sequential
  config := self initManagement
[] commsAgent := self initCommsAgent
[] userConnections := SmalltalkLibPkg:::Array new: maxConn
[] < [] i where 1≤i≤maxConn :: userConnections at: i
       put: (self initConnection: i)
  >
[] inputQ := SmalltalkLibPkg:::OrderedCollection new: maxConn
[] scAllocator := self initServiceCategoryAllocator
[] scContainer := SmalltalkLibPkg:::Array new: maxCategories
```

```
[] < [] j where 1≤j≤maxCategories :: scContainer at: j
     put: (CC_CorePkg:::ServiceCategory setup: config)
   >
[] spAgentContainer := SmalltalkLibPkg:::Array new: maxSP
[] < [] k where 1≤k≤maxSP :: spAgentContainer at: k
     put: (self initSPAgent)
   >
[] timer:= SystemUtilitiesPkg:::TimerServices setup: config
[] timerEventQ := SmalltalkLibPkg:::OrderedCollection new
end-sequential
```

The `initCommsAgent` method of the CC_Activation class contains a single statement indicating that the subclass needs to reimplement the method. It is therefore an **abstract** method. The **correctness property** of this method indicates that a non-nil value will be returned, but it does **not** specify **which** class should be instantiated.

```
message pattern initCommsAgent
method properties
"Total correctness"
true results-in methodReturnValue notNil                    "DL1-05"
sequential
self subclassResponsibility
end-sequential
```

CC_SimulationActivation, a subclass of CC_Activation, **reimplements** this method. In this case the **correctness property** specifies **explicitly** which class is instantiated:

```
message pattern initCommsAgent
method properties
"Total correctness"
true results-in
      methodReturnValue notNil ∧
      CC_SimulationInterfacesPkg:::CommsProviderSimulator
      postconditions: (#startSimulation)
                                        "DL1-05 (CC_Activation)"
sequential
^CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation
end-sequential
```

The Factory method may either be **abstract** or it may contain a **default implementation**. The `initialize` method of the CC_Activation class contains examples of both. The `initCommsAgent` and `initSPAgent` methods are both abstract, whereas the other Factory Methods contain default implementations. For example, the initServiceCategoryAllocator Factory Method instantiates the ServiceCategoryAllocator class by default:

```
message pattern initServiceCategoryAllocator
method properties
"Total correctness"
true results-in
      methodReturnValue notNil                              "DL1-07"
sequential
^CC_CorePkg:::ServiceCategoryAllocator setup
end-sequential
```

Default methods are convenient when a reasonable default exists. When a new system has to be built, the methods do not have to be overridden in subclasses if the default implementation suffices. However, should the default implementation not be adequate, the design provides the **flexibility** to override only the relevant Factory Method(s). For example, instead of having to

override the `initialize` method of the CC_Activation class (and thereby creating the risk of modifying unrelated code unintentionally), only the `initServiceCategoryAllocator` method needs to be reimplemented if a subclass of the ServiceCategoryAllocator class needs to be instantiated.

Abstract methods are used where a default implementation is not feasible. For example, the class representing the communication provider functionality is likely to differ amongst the various applications. A communication provider simulator may even be used, as in the example in this thesis. For the same reason, the class representing the service provider functionality is instantiated via an abstract method.

A number of varieties of the Factory Method pattern are described in [GHJV95]. One of the disadvantages of the Factory Method is the fact that the class which invokes the Factory Method (in this case CC_Activation) has to be subclassed when any of the classes which it instantiates need to be subclassed.

A variation of the Factory Method which **avoids subclassing** defines instance variables to hold the class names of all the classes that need to be instantiated by the creating class. Each Factory Method now creates an instance of a class by referring to the contents of one of these variables. This alternative, as applied to the CC_Activation example, is shown below:

The following instance variables are defined in addition to the ones already specified for the CC_Activation class in Appendix B, Section B.2:

```
managementClass
cpAgentClass
connectionClass
scAllocatorClass
spAgentClass
```

The `initialize` method is modified by including the following statement as the first statement in that method:

```
self makeClasses
```

The `makeClasses` private method is implemented as follows:

```
message pattern makeClasses
method properties
"Total correctness"
true results-in methodReturnValue = self ∧
      managementClass notNil ∧
      cpAgentClass notNil ∧
      connectionClass notNil ∧
      scAllocatorClass notNil ∧
      spAgentClass notNil

sequential
answerString :=
      Dialog[5] request: 'Call centre configuration class: '

[]managementClass :=
      Class readFrom: (ReadStream on: answerString)

[]answerString :=
      Dialog request: 'Communication provider agent class: '
```

[5] The Dialog class is used to request typed input from the user [HoHo95].

```
[]cpAgentClass := Class readFrom: (ReadStream on: answerString)

[]answerString := Dialog request: 'Connection class: '
[]connectionClass :=
      Class readFrom: (ReadStream on: answerString)

[]answerString :=
      Dialog request: 'Service category allocator class: '
[]scAllocatorClass :=
      Class readFrom: (ReadStream on: answerString)

[]answerString :=
      Dialog request: 'Service provider agent class: '
[]spAgentClass := Class readFrom: (ReadStream on: answerString)
end-sequential
```

Thus, for each class a dialog box is displayed requesting the name of the class to be instantiated. The string that is provided by the user is then used to create an object that is the required class name. This name is stored in the appropriate instance variable. The Factory Methods that instantiate the various classes are all modified to use the values in the instance variables containing the class names. Two examples are given below; the first is of a method that previously was an abstract method and the second example is of a method that contained a default implementation.

```
message pattern initCommsAgent
method properties
"Total correctness"
cpAgentClass notNil results-in methodReturnValue notNil"DL1-05"
sequential
^cpAgentClass setup
end-sequential

message pattern initServiceCategoryAllocator
method properties
"Total correctness"
scAllocatorClass notNil results-in
      methodReturnValue notNil                          "DL1-07"
sequential
^scAllocatorClass setup
end-sequential
```

Note that the precondition of each method now requires that the relevant instance variable containing the class name should contain a non-nil value. The **correctness properties** of the method still **do not refer explicitly** to the specific class that is instantiated. There is now **no need to subclass** the CC_Activation class in order to instantiate the appropriate communication provider and service category allocator classes.

If a design is used which employs instance variables to store the class names, then it would be prudent to use generic names such as `setup` for all the instance creation methods. A name such as `startSimulation` would therefore not be a good choice, since it implies that the object will be performing a simulation. However, if a class with such an instance creation method name already exists (as in the case of the CommsProviderSimulator class), one can always add a method called `setup` to the set of methods supported by the class. The `setup` method can then invoke the `startSimulation` method.

There are a number of reasons why the CC_Activation class might have to be subclassed. The application of the Factory method as described above enables one to eliminate the need for

subclassing when the CC_Activation class does not know in advance which **classes** will have to be **instantiated**.

Another reason why the CC_Activation class might have to be subclassed is the fact that it also does not know in advance which **parallel methods** need to be **activated**. The classes that are instantiated determine the parallel methods that have to be activated. If the client **interface** of the parallel methods remains the same for a class and its subclasses, the need for subclassing in order to activate the appropriate parallel methods is also eliminated. This aspect will be discussed in more detail in Section 9.5.3.

9.3.2 Singleton

The Singleton design pattern [GHJV95] is useful when it is necessary to restrict the number of instances of a class to one. The pattern allows this sole instance to be referenced globally without requiring the use of a global variable.

One possible implementation of the pattern is the following: The class that is designed as a Singleton has a class variable that contains the reference to the sole instance of that class. Whenever the instance has to be obtained, the instance creation method is invoked. The latter checks whether the class variable contains the value nil. If it does, a new instance is created and returned, otherwise the existing instance is returned.

When incorporating this design pattern into a SLOOP design an interesting question arises. Since the class that is implemented as a Singleton does **not necessarily** have to be **instantiated during activation**, a way has to be found to ensure that its **parallel statements are activated** if there are any associated with the class. This issue will be addressed shortly. First an example of a class designed as a Singleton is presented.

The Singleton design pattern is often used in conjunction with the Flyweight and State patterns [GHJV95]. These are described in detail in Sections 9.4.2 and 9.5.2 respectively. The State design pattern is used to extract state-specific behaviour from a class. A separate class is defined for each state of the original class. The latter then becomes the context of the state. The Flyweight pattern is used to make the instances of the state classes shareable by multiple instances of the context class. Only one instance of each state class is therefore required. The application of the Singleton design pattern ensures that only one instance is created for each state class.

When the State pattern is applied to the Connection[6] class of the call centre system, the IdleConnection, ConnectedConnection and TerminatingConnection classes emerge as state classes (the rationale for defining these particular classes is given in Section 9.5.2). The Connection class then becomes their context class. The application of the Flyweight pattern ensures that the state classes can be shared by multiple instances of the Connection class. Each state class can then be implemented as a Singleton.

The instance creation methods of the IdleConnection class demonstrate how the Singleton design pattern ensures that only one instance of the class is instantiated. The **new** method that is usually used for instance creation is overridden with the **instance** method as shown below:

```
class IdleConnection
superclass ConnectionState
class variable names
      IdleConnectionInstance
      "This variable is used to implement the class as a Singleton"
```

[6] The SLOOP specification of the Connection class is presented in Appendix B, Section B.7.

```
class properties
      invariant   IdleConnection instanceCount ≤ 1
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."

class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in methodReturnValue = IdleConnectionInstance ∧
            IdleConnectionInstance notNil
      sequential
      IdleConnectionInstance := super new
            if IdleConnectionInstance isNil
      [] ^IdleConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in IdleConnection postconditions: (#instance)
      sequential
      ^ IdleConnection instance
      end-sequential
```

When an invariant describes a property of an **instance** of a class, the invariant only needs to hold once the instance has been created and initialized. However, in the above example, the invariant describes a property of the **class**. This invariant has to hold at all times.

Instead of using a global variable to refer to the IdleConnection instance, all clients access the object via the message expression `IdleConnection instance`. This obviates the need to instantiate the IdleConnection class during initialization. The instance is created when it is used the first time.

As described above, state classes are often implemented as Singletons. Since many of these states are only reached after initialization has completed, the corresponding classes are only instantiated at that time. For example, the TerminatingConnection class, which is implemented as a Singleton, represents the behaviour of a connection when its state has changed to 'TERMINATING'. This class is only instantiated when the 'TERMINATING' state is entered for the first time.

Since the class is not necessarily instantiated during activation, it raises the issue of the **activation** of the parallel statements associated with the class, if there are any. The designer has to ensure that the requirements for SLOOP parallel statements are met at all times, i.e. all parallel statements required by the program have to be executed infinitely often. In order to facilitate reasoning about correctness, these statements have to be available for scheduling at all times after the instantiation of the class.

The only exception is when they are used in a very specific way, as described in Chapter 4, Section 4.3.5.3. To recapitulate: if a conditional expression is associated with a parallel statement, the statement has to be available for scheduling whenever the conditional expression evaluates to true. When the conditional expression evaluates to false, the statement need not be present, since the execution of the statement will have no effect even if it is present.

In the case of the Connection class, its parallel statements only have an effect if the Connection instance is in the 'TERMINATING' state, as can be seen from the cyclic methods listed in Appendix B, Section B.7. Thus, when these statements are moved to the TerminatingConnection class, it only needs to be guaranteed that these statements are available for scheduling **whenever** the Connection instance is in the 'TERMINATING' state, i.e. whenever the `state` instance variable of the Connection instance refers to the TerminatingConnection instance.

The statements in the `p_executeConnection:` method of the CC_Activation class are responsible for activating the parallel methods of the TerminatingConnection class, viz. the `p_informCommsProvider:context:` and `p_doWrapUp:` methods. In the statements below, these methods are only invoked if the connection is in the 'TERMINATING' state.

```
aConnection state p_informCommsProvider: commsAgent
context: aConnection
     if aConnection state = TerminatingConnection someInstance[7]
[] aConnection state p_doWrapUp: aConnection
     if aConnection state = TerminatingConnection someInstance
```

The parallel methods of the TerminatingConnection class are therefore invoked via the parallel statements in the *activation-section* of the program[8], but their invocation is subject to an instance of the TerminatingConnection class currently being in use by the relevant Connection instance. The **effect** that is achieved is therefore similar to that which applied when the statements were still part of the Connection class.

Thus, when the software designer incorporates a Singleton design pattern into a SLOOP design, it has to be done in such a way that the parallel statements that are involved can still be scheduled whenever they can have an effect. The way in which this can be achieved was demonstrated above.

Further clarification of the above example will be given in Section 9.5.2, where the State design pattern is explained in more detail.

9.4 Structural design patterns

9.4.1 Adapter

The communication provider and the service providers form part of the environment of the call centre system. The interfaces of these objects are therefore not determined by the designer of the call centre. These interfaces may vary, depending on the product being used. However, it is desirable to present a consistent interface to the clients of these objects within the call centre system. For this reason the CommsProviderAgent and ServiceProviderAgent classes are introduced. The clients within the call centre system deal with the agents only. The agents adapt the messages received from the clients to suit the interfaces of the actual communication and service providers being used. The agent classes are subclassed as needed, based on the interfaces of the communication and service provider classes. This is an example of an application of the Adapter design pattern. The latter is described in detail in [GHJV95].

In the SLOOP program given in Appendix B, the functionality of the agents is simulated via the CommsProviderSimulator and ServiceProviderSimulator classes. The interfaces presented to the communication and service provider clients are the same as those that should be presented by the corresponding CommsProviderAgent and ServiceProviderAgent classes. The latter may

[7] The someInstance method of the Smalltalk Behavior class returns an existing instance of the receiver.
[8] The p_activate message is sent to the CC_SimulationActivation instance from within the *activation-section* of the program. In turn, the p_activate method invokes the p_executeConnection: method inherited from the CC_Activation class.

therefore be defined as descendants of the respective simulation classes. Only the methods that perform the simulations need to be overridden.

There are no special considerations involved when applying this pattern to a SLOOP design.

9.4.2 Flyweight

One of the situations in which the Flyweight design pattern is applicable is when there is a proliferation of objects that may be reduced if some of these objects can be made shareable [GHJV95]. In order to use an object in multiple contexts simultaneously (i.e. as a Flyweight), it is necessary to store all information that is dependent on the context of the shared object in the context itself (this is called the extrinsic state of the Flyweight).

The Flyweight pattern is often used in conjunction with the State pattern. The latter implements the various states defined for a class as separate classes. The original class becomes the context of these state classes. All actions that are dependent on the state of the context are performed by the appropriate state objects. There is an instance of each state class for each instance of the context class. If the number of context instances is high and there are numerous states, then the application of the State design pattern can result in an unacceptably high number of objects. If the context-specific data can be stored in the context, then the instances of the state classes can be shared by multiple contexts, i.e. each state class can be implemented as a Flyweight.

This design pattern can be used successfully in a SLOOP design, provided that each parallel statement associated with the Flyweight is still executed infinitely often for each instance of the context class once the SLOOP program is **mapped** to an executable program. The way in which this can be achieved is discussed next.

The mapping of a SLOOP program to various architectures was discussed in Chapter 8. Each invocation of a parallel method results in the execution of one of its statements. The mapping algorithm has to guarantee that each statement of each activated parallel method will be executed infinitely often. One way of achieving this is by implementing an instance variable that keeps track of the last statement that was executed.

Since it has to be guaranteed that **each parallel statement** is executed infinitely often **for each context**, this variable (called the **statement selector) has to form part of the context**. The following scenario illustrates the problem that would occur if it were present in the state class instance (i.e. shared by multiple contexts).

Suppose there are four Connection instances and they are all in the same state, i.e. there are four context objects sharing a single instance of a state class. The latter has a single parallel method containing four statements. Suppose further that the parallel method is invoked for each Connection instance in a round robin fashion. This implies that during the first rotation the statements will be allocated to the various contexts as follows if the statement selector is shared by the contexts:

Context 1: statement 1
Context 2: statement 2
Context 3: statement 3
Context 4: statement 4

Unfortunately, the subsequent rotations will follow the same pattern, which means that for any given context, three of the statements will never be executed. It is therefore clear that the statement selector is context-specific and should be implemented as such. An example of the application of the Flyweight pattern in the call centre system is given in Section 9.5.2, where the State pattern is discussed.

9.4.3 Proxy

One of the applications of the Proxy design pattern is to allow one object to act as a local representative of a remote object [GHJV95]. It does not adapt one interface to another. This pattern is used extensively in distributed system infrastructures such as CORBA [BMRSS96]. Objects send messages to other objects without taking the location of the target objects into account, i.e. whether they are local or remote is irrelevant to the client object.

One possible implementation of such a system requires the instantiation of a proxy per address space for each object. The syntax of the messages sent to the proxy is exactly the same as for those sent to the original object. The proxy is responsible for obtaining information about the physical location of the original object and for performing the functions related to converting the message into a format that can be transmitted to a remote processor. These aspects are transparent to the client of the remote object.

This architecture is reused when SLOOP programs are **mapped** to distributed systems. As described in Chapter 8, SLOOP programs are designed in a unified manner. The physical location of each object is irrelevant. During the implementation phase the target architecture is determined. At that stage issues such as how to ensure the atomicity of SLOOP statements are addressed. Once a parallel statement has been selected for execution and all the required resources have been reserved for it, design patterns such as Proxy make it possible to send messages to local and remote objects within these statements without having to take their locations into account. Thus, the SLOOP program can be mapped to a distributed system without having to modify the design of the SLOOP program.

9.5 Behavioural design patterns

9.5.1 Iterator

The Iterator design pattern facilitates sequential access to the elements of a collection while hiding the underlying representation from the client [GHJV95]. Most class libraries provide this functionality for their collection classes. The Smalltalk internal Iterator method[9] `do:` is one example. This method evaluates the block that is supplied as argument to this method for each element of the collection representing the receiver of this message. The Iterator design pattern is therefore automatically used when SLOOP programs contain Smalltalk message expressions that invoke this method.

9.5.2 State

The Connection class of the call centre system maintains a state instance variable. If an event occurs, the behaviour depends on the current state of the object. The disadvantage of this design is the fact that the addition of a new state implies that each method that selects behaviour based on the current state of the object needs to be modified to include the behaviour for the new state.

The State design pattern [GHJV95] offers an alternative solution: the state-specific behaviour is removed from the original class. The latter becomes the **context** of a new class hierarchy. An abstract superclass is defined which represents the aspects that are generic to the states of the original class. Each state of the original class is implemented as a subclass of the new abstract

[9] Gamma et al. [GHJV95] define an internal iterator as one where the iterator controls the iteration. The client is responsible for advancing the traversal in the case of an external iterator.

class, therefore each method handling an event only contains the logic for the state represented by that particular class. The addition of a new state simply means the addition of a new subclass.

The following example shows how the **extensibility** of the Connection class is improved by using the State design pattern.

A new **abstract** class ConnectionState is defined, as shown in Figure 9-2. The Connection class becomes the **context** of the ConnectionState class. Three **concrete subclasses** are defined for the ConnectionState abstract class, viz. IdleConnection, ConnectedConnection and TerminatingConnection. They encapsulate the behaviour corresponding to the 'IDLE', 'CONNECTED' and 'TERMINATING' states previously defined within the Connection class. These classes are implemented as Flyweights, i.e. all the context-specific information is stored within the Connection class. (The application of the Flyweight design pattern was described in Section 9.4.2.)

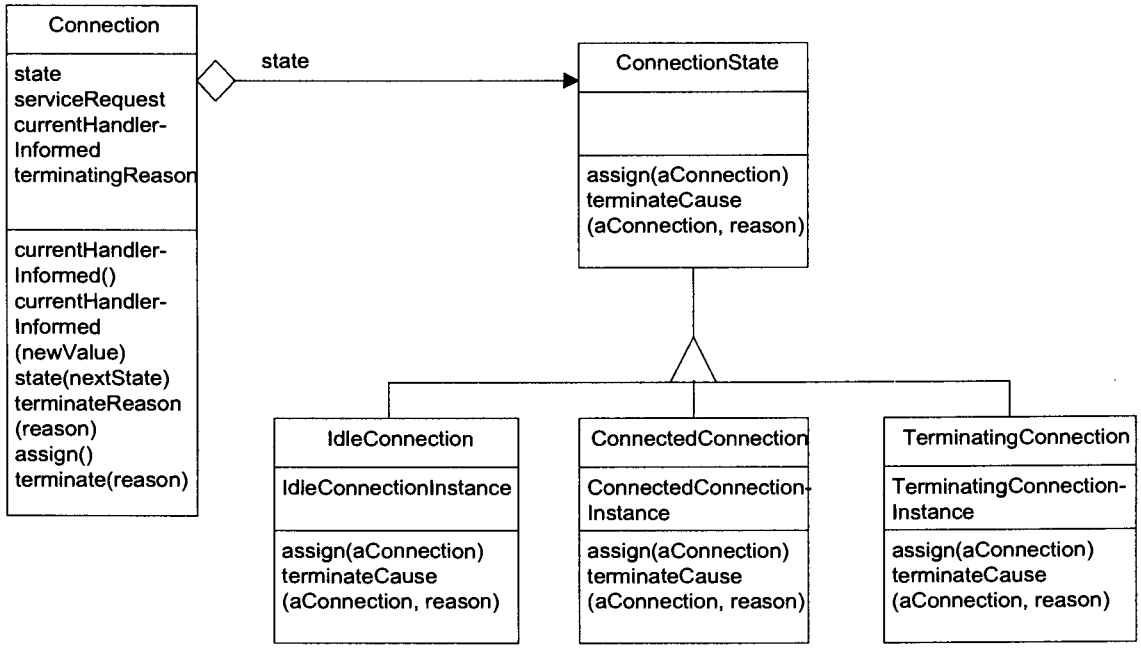

Figure 9-2. Incorporating the State design pattern into the Connection class.

The Connection instance still maintains an instance variable called state. However, instead of containing the values 'IDLE', 'CONNECTED' or 'TERMINATING', it now contains a reference to an instance of IdleConnection, ConnectedConnection or TerminatingConnection. When the Connection instance has to execute state-specific logic, it simply invokes the relevant method of the ConnectionState subclass instance that is currently referenced by its state instance variable.

The SLOOP specification of the original Connection class is given in Appendix B, Section B.7. The modifications to the methods of the Connection class are now presented (the modified parts are shown in bold italics):

```
category private
      message pattern initialize: indexOfConnection
      method properties
      "Total correctness"
      true results-in methodReturnValue = self ∧
      state = IdleConnection instance ∧
      serviceRequest notNil ∧ currentHandlerInformed = false ∧
      connectionIndex = indexOfConnection                          "DL1-11"
      sequential
      state := IdleConnection instance
      [] serviceRequest := CC_CorePkg:::ServiceRequest setup: self
      [] currentHandlerInformed := false
      [] connectionIndex := indexOfConnection
      end-sequential
```

When the Connection class is initialized, the state variable now contains a reference to an instance of the IdleConnection class. The method `instance` used here is a class method of the IdleConnection class. Since the latter is implemented as a Singleton (as described in Section 9.3.2), all accesses to the IdleConnection instance are via this method.

The next two methods are used by the clients of the Connection instance to determine whether the latter is idle or busy terminating. The Smalltalk `someInstance` method returns an existing instance of the receiver. Due to the application of the Singleton design pattern it is guaranteed that there will never be more than one instance of each of the ConnectionState subclasses. The statements in the methods below therefore suffice to provide the required answers.

```
category testing
      message pattern isIdle
      method properties
      "Total correctness"
      true results-in methodReturnValue =
            (state = IdleConnection someInstance)                  "DL1-05"
      sequential
      ^ state = IdleConnection someInstance
      end-sequential

      message pattern isTerminating
      method properties
      "Total correctness"
      true results-in methodReturnValue =
            (state = TerminatingConnection someInstance)           "DL1-06"
      sequential
      ^ state = TerminatingConnection someInstance
      end-sequential
```

Previously, accessing methods were provided to obtain the values of some of the instance variables of the Connection class, viz. `terminatingReason`, `serviceRequest` and `connectionIndex`. The values of the `state` and `currentHandlerInformed` instance variables were only significant to the Connection instance itself, therefore no accessing and modifying methods were provided for these variables. The introduction of the ConnectionState subclasses results in new accessing and modification methods being required, since actions that previously had been performed within the Connection class are now initiated externally (i.e. from within the ConnectionState subclasses). The new methods are shown next.

```
category accessing
      message pattern currentHandlerInformed
      method properties
      "Total correctness"
      true results-in methodReturnValue = currentHandlerInformed
```

```
        sequential
        ^currentHandlerInformed
        end-sequential

category modifying
        message pattern terminateReason: reason
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧
                terminateReason = reason
        sequential
        terminateReason := reason
        end-sequential

        message pattern state: nextState
        method properties
        "Total correctness"
        true results-in methodReturnValue = self ∧ state = nextState
        sequential
        state := nextState
        end-sequential

        message pattern currentHandlerInformed: newValue
        method properties
        "Total correctness"
        true results-in
        methodReturnValue = self ∧ currentHandlerInformed = newValue
        sequential
        currentHandlerInformed := newValue
        end-sequential
```

The methods that contain state-specific logic are the `assign` and `terminate:` methods in the modifying category, as well as the `p_informCommsProvider:` and `p_doWrapUp` methods in the cyclic category. The functionality of these methods is now delegated to the ConnectionState subclasses, as evident from the modified methods below:

```
category modifying
        message pattern assign
        method properties
        "Total correctness"
        state = IdleConnection someInstance results-in
                methodReturnValue = self ∧
                state postconditions: (#assign:) withArguments: #(self)
                                                                "DL1-07"
        sequential
        state assign: self
                if state = IdleConnection someInstance
        end-sequential

        message pattern terminate: reason
        method properties
        "Total correctness"
        state = ConnectedConnection someInstance results-in
                methodReturnValue = self ∧
                state postconditions: (#terminate:cause:)
                withArguments: #(self reason)                   "DL1-08"
        "Total correctness"
        state = TerminatingConnection someInstance results-in
                methodReturnValue = self                        "DL1-09"
        "This allows for terminate collision."
```

```
"Total correctness"
state = IdleConnection someInstance results-in
      methodReturnValue = self                          "DL1-10"
"This ensures that the transition from 'IDLE' to 'TERMINATING'
is not possible"
sequential
state terminate: self cause: reason
      if state = ConnectedConnection someInstance
end-sequential
```

The parallel methods are removed from the Connection class, since they are dependent on the state of the Connection instance.

The new ConnectionState class and its subclasses are now shown (the only class containing parallel methods is the TerminatingConnection class):

```
"---------------------------------------- class ConnectionState ---"
class ConnectionState
superclass Object
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      "The set of ConnectionState subclasses and the allowed state
      transitions are not specified here, because it would require
      ConnectionState to be subclassed if this was changed."
instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = IdleConnection someInstance results-in
            methodReturnValue = self ∧
            aConnection state postconditions: (#assign:)
            withArguments: #(aConnection)                "DL1-01"
      sequential
      self subclassResponsibility
      end-sequential

      message pattern terminate: aConnection cause: reason
      method properties
      aConnection state notNil results-in
            methodReturnValue = self ∧
            aConnection state postconditions: (#terminate:cause:)
            withArguments: #(aConnection reason)          "DL1-02"
      sequential
      self subclassResponsibility
      end-sequential
```

```
"---------------------------------------- class IdleConnection ---"
class IdleConnection
superclass ConnectionState
class variable names
      IdleConnectionInstance
      "This variable is used to implement the class as a Singleton"
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      invariant    IdleConnection instanceCount ≤ 1          "DS2-01"
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."
class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = IdleConnectionInstance ∧
            IdleConnectionInstance notNil                    "DL1-01"
      sequential
       IdleConnectionInstance := super new
            if IdleConnectionInstance isNil
      [] ^IdleConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in IdleConnection postconditions: (#instance)
                                                             "DL1-02"
      sequential
      ^ IdleConnection instance
      end-sequential

instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = IdleConnection someInstance results-in
            methodReturnValue = self ∧
            aConnection postconditions: (#state:)
            withArguments #((ConnectedConnection instance))
                                          "DL1-01(ConnectionState)"
      sequential
      aConnection state: (ConnectedConnection instance)
      end-sequential

      message pattern terminate: aConnection cause: reason
      method properties
      "Total correctness"
      aConnection state = IdleConnection someInstance results-in
            methodReturnValue = self       "DL1-02(ConnectionState)"
      "This ensures that the transition from 'IDLE' to 'TERMINATING'
      is not allowed"
      sequential
      ^self
      end-sequential
```

```
"------------------------------------ class ConnectedConnection ---"
class ConnectedConnection
superclass ConnectionState
class variable names
      ConnectedConnectionInstance
      "This is used to implement the class as a Singleton"
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      invariant   ConnectedConnection instanceCount ≤ 1      "DS2-01"
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."

class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = ConnectedConnectionInstance ∧
            ConnectedConnectionInstance notNil                "DL1-01"
      sequential
       ConnectedConnectionInstance := super new
            if ConnectedConnectionInstance isNil
      [] ^ConnectedConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in
      ConnectedConnection postconditions: (#instance)         "DL1-02"
      sequential
      ^ ConnectedConnection instance
      end-sequential

instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = ConnectedConnection someInstance results-in
            methodReturnValue = self       "DL1-01 (ConnectionState)"
      "This ensures that the transition from 'CONNECTED' to
      'CONNECTED' is not allowed"
      sequential
      ^self
      end-sequential

      message pattern terminate: aConnection cause: reason
      method properties
      "Total correctness"
      aConnection state = ConnectedConnection someInstance results-in
            methodReturnValue = self ∧
            aConnection postconditions: (#state:)
            withArguments: #((TerminatingConnection instance)) ∧
```

```
            aConnection postconditions: (#terminateReason:)
            withArguments: #(reason)        "DL1-02 (ConnectionState)"
      sequential
      aConnection terminateReason: reason
      [] aConnection state: (TerminatingConnection instance)
      end-sequential


"---------------------------------- class TerminatingConnection ---"
class TerminatingConnection
superclass ConnectionState
class variable names
      TerminatingConnectionInstance
      "This is used to implement the class as a Singleton"
instance variable names
      "There are no instance variables, which allows the class to be a
      Flyweight"
class properties
      invariant   TerminatingConnection instanceCount ≤ 1     "DS2-01"
      "The method instanceCount is a Smalltalk class method which
      returns the number of instances that currently exist for the
      specified class."

class methods
category instance creation
      message pattern instance
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = TerminatingConnectionInstance ∧
            TerminatingConnectionInstance notNil              "DL1-01"
      sequential
       TerminatingConnectionInstance := super new
            if TerminatingConnectionInstance isNil
      [] ^TerminatingConnectionInstance
      end-sequential

      message pattern new
      method properties
      "Total correctness"
      true results-in
      TerminatingConnection postconditions: (#instance)       "DL1-02"
      sequential
      ^ TerminatingConnection instance
      end-sequential

instance methods
category modifying
      message pattern assign: aConnection
      method properties
      "Total correctness"
      aConnection state = TerminatingConnection someInstance
            results-in methodReturnValue = self
                                          "DL1-01(ConnectionState)"
      "This ensures that the transition from 'TERMINATING' to
      'CONNECTED' is not allowed"
      sequential
      ^self
      end-sequential
```

```
        message pattern terminate: aConnection cause: reason
        method properties
        "This is the terminate collision case.  It is important not to
        overwrite terminateReason with reason, since the connection is
        already busy terminating."
        "Total correctness"
        aConnection state = TerminatingConnection someInstance
                results-in methodReturnValue = self
                                                "DL1-02(ConnectionState)"
        "This takes care of terminate collision."
        sequential
        ^self
        end-sequential

category cyclic
        message pattern p_informCommsProvider: commsAgent
                context: aConnection[10]
        method properties
        "Safe liveness"
        aConnection state = TerminatingConnection someInstance ∧
        aConnection terminatingReason = 'completed' ∧
        ¬(aConnection currentHandlerInformed) ensures
                commsAgent postconditions: (#terminate:cause:)
                withArguments:
                #(aConnection (aConnection terminatingReason)) ∧
                aConnection currentHandlerInformed              "DP1-01"
        parallel
        commsAgent terminate: aConnection
        cause: (aConnection terminatingReason) \+
        aConnection currentHandlerInformed: true
                if aConnection terminatingReason = 'completed'
                and: [aConnection currentHandlerInformed not]
        end-parallel

        message pattern p_doWrapUp: aConnection
        method properties
        "Safe liveness"
        aConnection currentHandlerInformed ensures
                aConnection state = IdleConnection someInstance ∧
                (aConnection serviceRequest) postconditions: (#reset) ∧
                ¬(aConnection currentHandlerInformed)           "DP1-02"
        parallel
        aConnection serviceRequest reset \+
        aConnection state: IdleConnection someInstance \+
        aConnection currentHandlerInformed: false
                if aConnection currentHandlerInformed
        end-parallel
```

The above implementation of the State design pattern provides an example of the usage of dynamic parallel statements.

In the original SLOOP program given in Appendix B, the Connection class contained two parallel methods, viz. `p_informCommsProvider:` and `p_doWrapUp`. These methods were invoked infinitely often for each Connection instance due to the presence of the following statements in the `p_activate` method of the CC_Activation class:

[10] The p_informCommsProvider: and p_doWrapUp methods are modified to include an additional argument, viz. the context.

```
[] < [] i where 1≤i≤maxConn ::
        self p_executeConnection: (userConnections at: i)
  >
```

The original p_executeConnection: method of the CC_Activation class contained the following statements:

```
"statements of the original p_executeConnection: method"
parallel
aConnection p_informCommsProvider: commsAgent
[] aConnection p_doWrapUp
end-parallel
```

The parallel statements of the Connection class only have an effect if the connection is in the 'TERMINATING' state, otherwise none of the **assignments** or **modifying** message expressions are executed (only the *if* clauses are executed). The parallel methods of the original Connection class are repeated here for easy reference. Those parts of the *if* clauses that refer to the state of the connection are highlighted in bold italics.

```
message pattern p_informCommsProvider: commsAgent
method properties
"Safe liveness"
state = 'TERMINATING' ∧ terminatingReason = 'completed' ∧
¬currentHandlerInformed ensures
        commsAgent postconditions: (#terminate:cause:)
        withArguments: #(self terminatingReason) ∧
        currentHandlerInformed                          "DP1-01"
parallel
commsAgent terminate: self cause: terminatingReason \+
currentHandlerInformed := true
        if state = 'TERMINATING' and:
        [(terminatingReason = 'completed')
        and: [currentHandlerInformed not]]
end-parallel

message pattern p_doWrapUp
method properties
"Safe liveness"
currentHandlerInformed ensures
state = 'IDLE' ∧ serviceRequest postconditions: (#reset) ∧
¬currentHandlerInformed                                 "DP1-02"
parallel
state := 'IDLE' \+
serviceRequest reset \+
currentHandlerInformed := false
        if currentHandlerInformed
"By following the logic of the methods of the Connection class
it will be evident that currentHandlerInformed can only be true
while the connection is in the 'TERMINATING' state"
end-parallel
```

When the State design pattern is implemented, the p_informCommsProvider: and p_doWrapUp methods are moved to the TerminatingConnection class. They are not present in the other State subclasses. Thus, when the connection is not in the 'TERMINATING' state (i.e. its state variable refers to a ConnectionState subclass instance other than the TerminatingConnection instance), the statements of these methods are not part of the list of parallel statements that are executed infinitely often.

The statements in the `p_executeConnection:` method of the CC_Activation class are changed to:

```
(aConnection state) p_informCommsProvider: commsAgent
context: aConnection
      if aConnection state = TerminatingConnection someInstance
[] aConnection state p_doWrapUp: aConnection
      if aConnection state = TerminatingConnection someInstance
```

Thus, the statements of the `p_informCommsProvider:context:` and `p_doWrapUp:` methods are present in the list of parallel statements whenever the Connection instance contains a reference to the TerminatingConnection instance (i.e. it is in the 'TERMINATING' state).

The statements of the `p_informCommsProvider:context:` and `p_doWrapUp:` methods no longer have to check the state of the Connection instance, since they can only be invoked if the TerminatingConnection instance is active.

This concludes the discussion of the State design pattern. The issues related to the SLOOP computational model were highlighted and it was shown why they did not present a problem. The above example has therefore demonstrated that the State design pattern can be applied successfully to a design based on the SLOOP method.

9.5.3 Template

The Template Method is used to implement the **skeleton of an algorithm**, allowing some steps to be reimplemented by subclasses [GHJV95]. This is achieved by invoking **abstract** methods or **default** methods for some steps of the algorithm. That way it ensures that all the necessary steps are executed and that they are performed in the correct order, while allowing subclasses to redefine the variant parts of the algorithm.

In Section 9.3.1 it was mentioned that the `initialize` method of the CC_Activation[11] class was implemented as a Template Method. In that section the focus was on the Factory Methods that were used as abstract or default methods. The discussion here concentrates on the Template Method characteristics of the design of the `initialize` method.

The method includes all the statements that are necessary to instantiate all the classes required by the system. The order in which these statements appear ensures that the correctness properties specified for the method are satisfied. For example, the `config` object (created via the `initManagement` method) has to exist prior to the instantiation of many of the other classes, since configuration information is obtained from the `config` object during the instantiation of these classes. By reusing the `initialize` method, the designer is assured of performing the instantiation actions in the correct order.

The *activation-section* of a SLOOP program not only contains statements to instantiate the relevant classes, but it also **activates** the appropriate **parallel statements**. These parallel statements may vary, depending on the classes that are instantiated. The Template Method design pattern is therefore also used in the `p_activate` method of the CC_Activation example. It contains the following statements:

[11] The CC_Activation class is specified in Appendix B, Section B.2.

```
parallel
self p_executeCPAgent
"Execute the parallel statements of the commsAgent."

[] timer p_runTimer: timerEventQ
"Whenever a timeout occurs, the TimeoutElement instance
representing the timeout is added to the end of the timerEventQ,
which indicates to the requestor that the specified timer has
expired."

[] self p_categoriseAndAllocate
"Once a service request has been categorised, it is removed from
the inputQ and appended to the appropriate serviceQ."

[] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
      p_execute
  >
"For each service category the associated service queue and set
of service provider agents are monitored.  If the service queue
is not empty and a service provider agent in the spSubset
associated with the service category is available to process a
new service request, the first element of the service queue is
removed and assigned to a service provider agent."

[] < [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
  >
"When a connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection.  Once all the procedures have been completed to
terminate the connection, the connection and its associated
service request are reset to their initial states."

[] < [] k where 1≤k≤maxSP :: self p_executeSPAgent:
      (spAgentContainer at: k)
  >
"Execute the parallel statements of the service provider
agents."

end-parallel
```

As mentioned earlier, the Template Method may invoke abstract or default methods. The example above illustrates both types of invocations. The `p_executeCPAgent` method is abstract, whereas the `p_categoriseAndAllocate` method is a default method. The implementation of each method is shown below:

```
message pattern p_executeCPAgent
method properties
"These are the properties pertaining to the communication
provider interface as identified during the analysis phase."
parallel
self subclassResponsibility
end-parallel

message pattern p_categoriseAndAllocate
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
```

```
parallel
scAllocator p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] scAllocator p_allocate: scContainer from: inputQ
"Once the service request has been categorised, the scAllocator
removes it from the inputQ and appends it to the appropriate
serviceQ."
end-parallel
```

By using the Template Method design pattern, it is clear to the designer of a new application that the parallel statements of the `cpAgent` object should be included, but the actual statements are only specified once the relevant interface class is determined. In the case of the `p_categoriseAndAllocate` method a default implementation is feasible, which is provided for the convenience of the designers of future applications.

As described in Section 9.3.1, subclassing can be avoided when using certain variants of the Factory method during instance creation. This is only beneficial if the parallel methods belonging to the classes being instantiated can also be activated without having to resort to subclassing. This implies that although the set of parallel methods may differ for each class, they have to be activated via the same message expression. One way of achieving this is by **encapsulating** the parallel methods of each subclass in such a way that **all subclasses present the same interface** to the client.

For example, a subclass of the ServiceCategoryAllocator[12] class might obtain information from a database in order to categorise a service request. In order to accomplish this, it may be necessary to define additional parallel methods for the ServiceCategoryAllocator subclass as well as modify the ones inherited from the parent class. These changes can be hidden from the client if the parallel methods of the ServiceCategoryAllocator class and its subclasses are encapsulated in a new method, viz. `p_categorise:allocate:`

The `p_categoriseAndAllocate` method of the CC_Activation class now only refers to this encapsulating method, as shown below:

```
message pattern p_categoriseAndAllocate
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
parallel
scAllocator p_categorise: inputQ allocate: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request).  Once it has been categorised, it removes it from the
inputQ and appends it to the appropriate serviceQ."
end-parallel
```

The `p_categorise:allocate:` method of the **ServiceCategoryAllocator** class now invokes the methods previously invoked by the `p_categoriseAndAllocate` method of the **CC_Activation** class:

```
message pattern p_categorise: inputQ
      allocate: scContainer
```

[12] The ServiceCategoryAllocator class is specified in Appendix B, Section B.8.

```
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
parallel
self p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] self p_allocate: scContainer from: inputQ
"Once the service request has been categorised, the scAllocator
removes it from the inputQ and appends it to the appropriate
serviceQ."
end-parallel
```

Subclasses of the ServiceCategoryAllocator class may alter the implementation of the `p_categorise:allocate:` method (e.g. by adding parallel statements to obtain information from a database) without affecting the CC_Activation class.

Note that the encapsulating method has to pass all the arguments required by the encapsulated parallel statements. This interface remains the same, even though some subclasses may not require all the arguments.

9.5.4 Strategy

Before a service request can be allocated to one of the call centre service queues, its category has to be determined. Different applications may require different categorisation algorithms to be used. For example, the information could be extracted from the service request itself, a database could be consulted or the call centre could enter into a dialogue with the service user to obtain the information. The Strategy pattern is useful to allow one to vary the algorithm without requiring subclassing.

The `categoriseServiceRequest:using:` method of the ServiceCategoryAllocator class implements the categorisation algorithm in the call centre system. In the original SLOOP program given in Appendix B, the ServiceCategoryAllocator class needs to be subclassed if the default implementation of this method presented in Section B.8 does not suffice.

Incorporating the Strategy pattern involves the definition of a new class, viz. CategorisingStrategy. It represents the common interface used by the context when the latter invokes one of the supported algorithms. The subclasses of this class represent the various options as listed above, as well as any future implementations. The abstract class is defined as follows:

```
class CategorisingStrategy
superclass Object
instance methods
category modifying
      message pattern categoriseServiceRequest: serviceRequest
                        using: scContainer
      method properties
      "Total correctness"
      "When this method has completed execution, the
      serviceRequestCategory attribute of the service request object
      will have a value (i.e. the service request will have been
      categorised) and that category will match one of the service
      categories supported by the system."
      true results-in
            methodReturnValue = self ∧
```

```
            serviceRequest serviceRequestCategory notNil ∧
            (scContainer detects:
            ([:each | each serviceQCategory =
            serviceRequest serviceRequestCategory] ifNone: [nil]))
            notNil                    "DL1-01 (CategorisingStrategy)"
      sequential
      self subclassResponsibility
      end-sequential
```

The DefaultCategory subclass is presented next:

```
class DefaultCategory
superclass CategorisingStrategy
instance methods
category modifying
      message pattern categoriseServiceRequest: serviceRequest
                        using: scContainer
      method properties
      "Total correctness"
      true results-in
            methodReturnValue = self ∧
            serviceRequest serviceRequestCategory notNil ∧
            (scContainer detects:
            ([:each | each serviceQCategory =
            serviceRequest serviceRequestCategory] ifNone: [nil]))
            notNil                    "DL1-01(CategorisingStrategy)"
      sequential
      serviceRequest serviceRequestCategory:
      (scContainer first) serviceQCategory
      end-sequential
```

The `p_categorise:using:` method of the **original** ServiceCategoryAllocator class contains the following statement to invoke its own `categoriseServiceRequest:using:` method:

```
      parallel
      categorising := true \+
      self categoriseServiceRequest: (inputQ first) using: scContainer
            if inputQ isEmpty not and: [categorising not]
      end-parallel
```

This method is now modified to invoke the method of the appropriate CategorisingStrategy subclass, as shown below. The new `categorisingAlgorithm` instance variable of the ServiceCategoryAllocator class contains a reference to the instance of the relevant CategorisingStrategy subclass. The way in which this variable is initialized will be explained shortly. The method properties of the modified `p_categorise:using:` method also reflect the new receiver of the `categoriseServiceRequest:using:` message.

```
category cyclic
      message pattern p_categorise: inputQ using: scContainer
      method properties
      "Safe liveness"
      ¬(inputQ isEmpty) ∧ ¬categorising until
            categorising ∧
            categorisingAlgorithm postconditions:
            (#categoriseServiceRequest:using:)
            withArguments: #((inputQ first) scContainer)
                              "DP1-03 (ServiceCategoryAllocator)"
```

```
parallel
categorising := true \+
categorisingAlgorithm categoriseServiceRequest: (inputQ first)
using: scContainer
      if inputQ isEmpty not and: [categorising not]
end-parallel
```

The `categorisingAlgorithm` variable is initialized as follows: When the ServiceCategoryAllocator instance creation method is invoked, the name of the appropriate CategorisingStrategy subclass is passed as an argument. During initialization of the ServiceCategoryAllocator instance, the CategorisingStrategy subclass instance is created and the reference is stored in the `categorisingAlgorithm` variable.

If the CategorisingStrategy subclasses contain parallel methods, then the statements in these methods have to be activated. A **common interface** should be defined for the invocation of the parallel methods of the various subclasses. That would facilitate the activation of the parallel statements of the instantiated subclass by merely **adding a statement to invoke this common method** to the `p_categorise:allocate:` method. The latter is the method which invokes all the parallel methods of the context. It was first introduced in Section 9.5.3. Should the CategorisingStrategy subclasses contain parallel methods, the required modification would be as shown below in bold italics:

```
message pattern p_categorise: inputQ
      allocate: scContainer
method properties
"These are the properties pertaining to the service category
allocator as identified during the analysis phase."
parallel
self p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] self p_allocate: scContainer from: inputQ
"Once the service request has been categorised, the scAllocator
removes it from the inputQ and appends it to the appropriate
serviceQ."
[] categorisingAlgorithm p_execute
"Execute the parallel statements of the CategorisingStrategy
subclass."
end-parallel
```

From the above it is clear that the Strategy pattern is most suited to a design where such common interfaces can be defined, otherwise it would be more appropriate to subclass the original class.

9.6 Summary

The SLOOP method differs from more conventional object-oriented approaches in the sense that it is based on a **different computational model**. In previous chapters the advantages of this method were described, e.g. its high level of abstraction, its applicability to all types of architectures and its emphasis on correctness properties. In this chapter it was demonstrated that the computational model of the SLOOP method presented no difficulties when applied to such a wide variety of design problems as exemplified by those given in [GHJV95] and [BMRSS96].

Several patterns were incorporated into a design based on the SLOOP method. Each category described in [GHJV95] and [BMRSS] was covered. The suitability of the SLOOP method was discussed for multiple design patterns in each category. The results can be summarized as follows:

- The structure of the **Pipes and Filters architectural pattern** [BMRSS96] promotes parallelism. The filters can be implemented to **execute concurrently** and to be **non-terminating**. These characteristics are **inherent** in the SLOOP approach, since the latter is based on the concept of a number of parallel statements that execute infinitely often.

- The **Reflection architectural pattern** [BMRSS96] was applied during the SLOOP implementation phase. In Chapter 8 it was shown how it could be used to **control statement execution**, perform some **assertion checking** and **generate trace information**.

- The **Factory Method creational design pattern** [GHJV95] allows a class to **defer** the specification of exactly **which** classes to instantiate to its subclasses. This design pattern is easily incorporated into a SLOOP design. Variants that **avoid subclassing** can also be used, but it was pointed out in Section 9.3.1 that this goal can only be achieved successfully if the **parallel** methods of the relevant classes could also be **activated** without requiring subclassing. This is possible if the **client interface** of the parallel methods can be defined to **remain the same for a class and its subclasses**.

- The **Singleton creational design pattern** highlighted another aspect of parallel statement activation. **All the parallel statements** required by a particular application have to be activated via the parallel statements in the *activation-section* of the SLOOP program. However, the instances of some classes may not yet exist immediately after the sequential statements in the *activation-section* have been executed. For example, the class which represents the 'TERMINATING' state of a connection is only instantiated once this state is entered for the first time. The invocation of the parallel methods of such a class therefore has to be **subject to the existence of an instance** of that class. As explained in Section 9.3.2, this is only acceptable if the **netto effect** of the execution of the affected parallel statements is the **same before and after incorporating the design pattern**. An example of how this could be achieved was presented in Section 9.3.2.

- When the **Adapter structural design pattern** is used in a SLOOP design, **no special considerations** are necessary.

- In contrast, care has to be taken that the **mapping** of the collaborators in the **Flyweight structural design pattern** is performed correctly during the implementation phase. A statement selector has to be maintained for **each context** of the shared object in order to guarantee that each parallel statement of the Flyweight will be executed infinitely often for each context.

- One of the applications of the **Proxy structural design pattern** is to make the **physical location** of objects **transparent** to the application. This is in line with the philosophy used in the SLOOP method, which advocates a **unified design approach,** i.e. the target architecture is only considered during the implementation phase.

- The **Iterator behavioural design pattern** is present in Smalltalk library classes. Since **Smalltalk message expressions** may form **part of SLOOP statements**, this design pattern is used extensively in most SLOOP programs.

- The **State behavioural design pattern**, discussed in Section 9.5.2, provided an example of the use of **dynamic** parallel statements. Prior to the incorporation of this design pattern, the parallel statements of the Connection class were **always present in the list of parallel statements**, but they only had an **effect** when the connection was in the 'TERMINATING' state. When these statements were moved to the TerminatingConnection class, they were only **present** in the list of executable statements when the state instance variable of the Connection class referred to the TerminatingConnection class, i.e. when the connection was

in the 'TERMINATING' state. When the application of the State design pattern results in the use of dynamic parallel statements, the designer has to ensure that the **effective** behaviour is the same before and after the incorporation of the pattern.

- The **Template Method behavioural design pattern** is very useful in a SLOOP program. It provides **flexibility** while at the same time it enables the designer to ensure that the **relevant statements will be executed**. This applies to both **sequential and parallel** Template Methods. If the method is **sequential** and **correctness properties** are specified that refer to the **ordering** of the statements, the Template Method allows one to guarantee that these properties will not be violated. Subclasses may only change the **contents** of the methods invoked by the Template Method, but not the invocations themselves.

- In order to use a Template Method design pattern for the invocation of parallel methods, the **client interface** of these methods has to be **the same for a class and its subclasses**. A similar requirement exists when the **Strategy behavioural design pattern** is applied to a SLOOP design.

This chapter concludes the presentation of the various aspects of the SLOOP method. The next chapter summarises the advantages of using this method and it also describes the directions for future research.

CHAPTER 10

CONCLUSIONS

10.1 Evaluation of the SLOOP method

The preceding chapters described all facets of the SLOOP method, viz.

- its syntax,
- the associated semantics,
- the analysis and design approach that results from applying the method,
- reasoning about correctness properties on an informal basis,
- the mapping of a design to an executable program,
- the use of reflection to separate the statements **within** the system being designed from the statements **about** the system being designed, and
- considerations when incorporating various design patterns into SLOOP designs.

Elaborate examples demonstrated various aspects of the SLOOP method. It is now appropriate to evaluate the SLOOP method with respect to the goals of this research as listed in Chapter 1.

10.1.1 Increasing the reliability of systems developed via this method

The first goal, viz. to **maximise** the **reliability** of the system under development, encompasses a wide spectrum of issues. First of all, the system that is produced has to be **functionally correct**. In order to achieve this, the SLOOP method requires the software designer to focus on correctness properties throughout the system development.

During the analysis phase, the **behaviour of the system** is specified in terms of a set of **informal correctness properties,** once the interacting classes have been identified. The SLOOP method aids the designer by providing a **useful checklist**[1] of different kinds of correctness properties that can be specified. This prompts the designer to analyse the problem domain in terms of a wider range of aspects than might otherwise have been the case. This checklist therefore promotes the **completeness** of the specification.

During the design phase, the focus remains on the correctness properties, but now they are **also** used to find suitable **matching** artifacts in the repository of **reusable artifacts** (if one exists). Artifacts are therefore compared on the basis of their correctness properties as was shown in Chapter 6. During the design phase, the correctness properties are **refined**. They are also specified more rigorously in order to facilitate an **unambiguous** specification, which is required in order to **reason about the correctness** of the design. The SLOOP method provides a **notation**[2] based on temporal logic for the rigorous specification of correctness properties.

---

[1] This checklist of correctness properties was described in detail in Chapter 5, Section 5.2.4.
[2] The SLOOP notation for specifying correctness properties was presented in Chapter 4, Section 4.3.4.

During the implementation phase, the target architecture is considered for the first time. When the SLOOP program is mapped to an executable program, the designer has to take care that the semantics of the SLOOP statements are preserved. The issues that need to be taken into account in order to achieve this, were discussed in Chapter 8. The SLOOP method advocates the development of **infrastructures** for mappings to different types of architectures (as described in Chapter 8). The correctness properties of these infrastructures can then also be reused.

It is evident from the above that the SLOOP method provides several mechanisms in order to aid the achievement of **functional** correctness during system development. Functional correctness is a requirement for all types of architectures.

The second aspect of the goal of producing reliable systems is to ensure that the problems usually associated with **concurrency**, such as **deadlock**[3] and **interference**[4], are prevented. Since the SLOOP method advocates a **unified** approach towards software development, these are issues that are only relevant during the implementation phase. The system is designed at a **high level of abstraction**. By definition, the unit of atomic execution in a SLOOP program is a parallel statement. At the design level, all the actions that should take place **atomically**, are grouped into a single parallel statement. There can be no interference between parallel statements.

As described in Chapter 4, Section 4.3.6.4, the SLOOP method is based on the **interleaving model of concurrency** [MaPn81a]. Thus, if two parallel statements refer to the same objects, they execute in some arbitrary order; if they do not share any objects, they may execute simultaneously. During the implementation phase the atomicity of the parallel statements has to be preserved in order to prevent interference. Furthermore, the interleaving model has to be preserved in order to ensure the prevention of deadlock. Chapter 8 covered possible strategies to achieve this.

As was evident from the earlier chapters, deadlock and interference are not the only issues addressed by the SLOOP method. Many different types of **safety**, **liveness** and **precedence** properties are described. Although the software designer is only required to reason about these properties informally, the mere fact that the **"constructive approach"** [Meye90] is followed during system development results in a product that instills more confidence as far as its correctness is concerned. As was demonstrated in Chapter 7, the SLOOP method encourages the software designer to consider both what should happen and also what should never happen. The end result is a more reliable system. This was corroborated by the results of the experimental systems that were developed.

In [Meye97] Bertrand Meyer states that "it is still too difficult to produce software without defects (bugs), and too hard to correct the defects once they are there." He continues to list some of the techniques for improving the reliability of software. These are, *inter alia*, a more **systematic** approach towards software construction, **more formal specifications** and **built-in checks** throughout the software development process.

The SLOOP method applies all of these techniques: It provides a checklist of useful correctness properties, thereby encouraging the software designer to work more **systematically**. The behaviour of the system has to be specified in terms of a set of correctness properties. The SLOOP notation provides the necessary constructs to express these properties **formally**. Although the correctness arguments are informal, they form an integral part of the method, which attests to the significance attached to them. Each phase of the software development process

[3] The conditions for deadlock to occur, as well as deadlock prevention strategies, were described in Chapter 4, Section 4.3.6.5.
[4] Interference was defined in Chapter 2, Section 2.3.2.

emphasizes correctness. By using the SLOOP method, the software developer therefore automatically focusses on correctness issues **throughout** the software development process.

10.1.2 Scalability of the method

When the scalability of the SLOOP method comes under scrutiny, one can argue that there are two aspects that need to be considered when the method is applied to medium- to large-scale systems. The one aspect deals with the software lifecycle in general, i.e. the mechanisms that are provided by the method to handle the **analysis, design and implementation phases**. The second aspect deals with how well the method facilitates **reasoning about correctness**.

When evaluating the way in which the SLOOP method assists the software designer during the analysis, design and implementation phases, the following is apparent. Since the SLOOP method is an object-oriented method, it has all the **structuring capabilities** that are associated with **object-orientation**. Thus, the solution domain is modelled in terms of a set of classes. The SLOOP method takes full advantage of the **data encapsulation** feature of object-orientation. Even the **parallel statements** are defined on a per class basis and are **encapsulated** within parallel methods associated with specific classes. The SLOOP method therefore provides the software designer with the necessary structuring mechanisms in order to break a large system into smaller, more manageable components.

As far as reasoning about correctness is concerned, the SLOOP method has several features that simplify correctness arguments. The benefits are particularly noticeable in larger systems. As demonstrated in Chapter 7, the **correctness arguments do not refer to location counters**. This is because the properties are existentially or universally qualified over all program statements. In larger systems this **reduces the complexity** of the correctness arguments considerably, since in a SLOOP program there is **no need to take computation histories into account**. In a system with a conventional computational model, the number of computation histories grows exponentially as the number of processes that are involved increases.

Location counters are only significant in **sequential methods** in SLOOP programs, but since a sequential method is always executed as an **atomic** unit, there can be no interference and the correctness arguments are therefore as for a sequential program, i.e. relatively simple. Since a sequential method is typically a very **small** piece of code, the software designer only has to deal with a small piece of logic at a time.

Although the correctness properties are quantified over **all** the parallel statements of a SLOOP program, it is typically **only a few** of these statements that actually influence a specific property. Only those parallel statements that reference the objects mentioned in the correctness properties, and possibly a few related ones, need to be taken into account. This was demonstrated in many examples in Chapter 7. Thus, although it might seem that the size of a system would adversely affect the ease with which one could reason about correctness, this is not the case because the **parallel statements** of the system **do not have a flat structure**. Parallel statements are **encapsulated** within the **parallel methods of objects** and are therefore structured according to the classes of the system. Furthermore, the parallel methods of the classes have **correctness properties** associated with them that specify clearly the effects of executing the statements contained within them.

Another feature of the SLOOP method which makes it particularly appropriate for larger systems, is its capacity for **reuse**. Not only can designs and code be reused, but **correctness reasoning** also does **not** need to take place **from first principles** each time. This was demonstrated in numerous examples in Chapter 7. Furthermore, the `postconditions:` and `postconditions:withArguments:` constructs in the **SLOOP notation** make it possible to

highlight the fact that other methods are being invoked and that their correctness properties are being **reused**.

The running example that was used in the body of this thesis was specifically chosen so that the applicability of the SLOOP method to **non-trivial** systems could be demonstrated.

10.1.3 Understandability of the method

A number of issues affect the understandability of the SLOOP method. First of all, the underlying **computational model** has to be understood by the software designer. The user of the SLOOP method has to think in terms of statements that execute infinitely often. Although this is different from conventional computational models, the principle is simple.

The second aspect that a new user of the SLOOP method has to grasp, is how **object-orientation** fits into the picture. Again, this is not complex. The (static) object model of the system is created in the usual way. It is only once the **behaviour** of the system is specified that the computational model has an effect. The designer has to determine how the functionality of a class should be distributed amongst its **sequential** and **parallel** methods. The designer also has to determine which actions should be executed **atomically**, i.e. which actions should be grouped into a single parallel statement.

The fact that the behaviour of the system and its classes is first specified via a set of properties (as was shown in Chapters 5 and 6), makes it relatively easy to derive the SLOOP statements. This is because one has a clear specification of what the behaviour of each class and its methods should be.

One of the main criticisms against formal methods is the perception that the underlying mathematics is difficult and tedious to use. Although the SLOOP method is not a formal method, a certain amount of rigour is required in order to support correctness reasoning. The underlying mathematical foundation for the SLOOP method is based on UNITY [ChMi88]. In [GePn89] a proof system is given for UNITY. It is claimed there that the UNITY assignments could be substituted with arbitrary programs without having to change the existing rules of the proof system; only a few additional rules would have to be added to reason about these atomic programs. This forms the theoretical basis for allowing method invocations in SLOOP statements.

However, **the user is not required to have any detailed knowledge of the theory underpinning the SLOOP method**; not even when reasoning about the correctness of a SLOOP program. As was demonstrated in Chapter 7, the **correctness arguments** are **informal**. There is no need to learn a set of theorems and to understand the application of such theorems. A correctness property is shown to be correct by inspecting the other correctness properties of the program and using them in the correctness arguments. If there are no relevant properties, the SLOOP statements themselves are inspected in order to support the correctness arguments.

Several factors contribute to ensure that this is not such a daunting task. First of all, there is **no need to take location counters into consideration**. Secondly, the **structuring** and the **data encapsulation** provided by object-orientation, results in the **localisation** of the properties (and statements) that need to be considered. It is seldom that all the properties specified for a program need to inspected in order to reason about the correctness of a specific aspect. Furthermore, each correctness property is only proved **once from first principles**. Thereafter its results can be **reused** in other correctness arguments. This reusability contributes towards making the proofs less tedious.

Another important aspect which simplifies the correctness properties and therefore aids understandability, is the fact that **class and instance methods may be used in the correctness**

properties. The only proviso is that they should not modify the state of the objects. One is therefore not restricted to specifying the properties in terms of class and instance variables and boolean operators. The **expressive power** gained from allowing methods in the correctness properties contributes towards making the specification of these properties **less tedious** and it also makes them **more understandable**. It enables one to write these properties in terms of methods rather than in terms of variables, i.e. they are expressed at a **higher level of abstraction**.

In the Eiffel programming language, method invocations are also allowed in the assertions [Meye97]. However, the SLOOP method goes even further. The notation contains the **`postconditions:` and `postconditions:withArguments:` constructs** that enable one to specify which other modifying methods are invoked by the method currently under consideration. This makes it **explicit** that another method is involved. It makes it clear that the postconditions of the other method will hold, without stating what those postconditions are. This **minimises the risk of inconsistency**, since the actual postconditions of that method are specified at one location only, viz. where that method is defined.

Finally, the SLOOP syntax includes **Smalltalk message expressions**, which makes SLOOP programs easily understood by software designers already familiar with Smalltalk.

10.1.4 Unified approach

The SLOOP method is not concerned with the target architecture during the analysis and design phases of system development. This was demonstrated in the example used in Chapters 5 and 6. The **target architecture** is only considered once the **implementation phase** is reached. In Chapter 8 it was shown what issues needed to be considered when a SLOOP program was mapped to various architectures. In all cases the **semantics** of the SLOOP statements, as well as the **atomicity** of the SLOOP parallel statements had to be preserved. This was achieved in different ways for the various architectures, as described in Chapter 8.

Although the mapping to a specific architecture is an additional step that is not required in a software development method where the program is designed for a specific architecture from the outset, a unified approach provides the software designer with the **freedom to map the design to any target architecture** with relative ease. In our experience, there has not been the need to make any modifications to the design as a result of a mapping to a particular target architecture.

A unified approach also has the advantage that the design is at a **high level of abstraction**. There is no need to consider issues such as mutual exclusion and deadlock that would normally be associated with concurrency. Those aspects are addressed during the implementation phase. As shown in Chapter 8, the solutions that are implemented for the various target architectures during that phase can be **reused in infrastructures** for the respective target architectures.

10.1.5 Reusability

The SLOOP method has all the **reusability features** of a **fully-fledged object-oriented method**. In Chapter 9 it was demonstrated how **design patterns** could be reused in a SLOOP design. In addition to the usual reuse of classes and patterns, the SLOOP method also makes provision for the reuse of **correctness properties**, as was described in Chapter 7. During the implementation phase, the **mapping infrastructures** can also be reused. This was discussed in detail in Chapter 8.

10.1.6 Seamlessness

During the analysis phase, the behaviour of the system under development is described in terms of a set of correctness properties. During the design phase, these properties are refined and

additional design level properties are added. The notation used for the specification of the properties includes Smalltalk message expressions. This makes the derivation of the SLOOP statements from the correctness properties relatively simple. If the target implementation language is Smalltalk-80, the transition from the design phase to the implementation phase is **seamless**.

As demonstrated in Chapter 8, it is possible to use the **reflective facilities** of Smalltalk to ensure that the base objects do not contain any additional logic that are specific to the mapping of the program to the target architecture. For example, the statements related to the selection of the next parallel statement for execution is relegated to the metaclasses. The mapped base class and the original SLOOP class are therefore almost identical, thereby achieving a high degree of seamlessness.

10.1.7 General availability and minimisation of developmental resources

At this stage a SLOOP development environment does not exist yet. Currently, a SLOOP program can be written using any text editor. When the mapping is performed, the development environment of the target architecture is used. For example, for a mapping to a sequential architecture the Smalltalk-80 development environment can be used as is. This also applies to a mapping to an asynchronous shared-memory architecture where multiple processes run on a single processor. The meta-object infrastructure used for reflective computation and the mapping infrastructures are implemented as reusable classes. All of these factors make it easy to experiment with the concepts proposed in the SLOOP method without having to make a large investment in terms of developmental resources.

10.2 Concluding remarks and future research directions

The purpose of this research has been to **experiment** with the concept of an **object-oriented method based on a Single Location Program (SLP) computational model** in order to try and achieve the goals discussed in the previous section. As recorded in the preceding chapters and summarised in the previous section, the SLOOP method provides the necessary features to accomplish this.

This experiment has therefore **resulted** in the development of a **new software construction method** which has the rich feature set of an object-oriented method, but which is based on a computational model that simplifies correctness reasoning. This **simplification**, which is enhanced by the high degree of **reuse** that is facilitated by the object-oriented nature of the method, has the following implications:

- It improves the understandability of the method.
- It makes the specification of correctness properties a simpler and less tedious task.
- Informal correctness reasoning about these properties becomes viable.
- It makes the method attractive to practising software designers that are not necessarily proficient in the use of formal methods.

The SLOOP method therefore promotes a **"constructive approach"** towards software development.

Although the emphasis in this research has been on creating a software development which facilitates **informal** correctness reasoning, this could be **complemented** by the development of a formal proof system for the SLOOP method. That way, the method would still be usable without requiring any knowledge of formal methods if only informal correctness arguments were used, but the user would also have the option of creating formal proofs. **Further research** would be required to develop a **formal semantics and proof system** for the SLOOP method. That would

then also facilitate the development of **tools** to **automate** the verification of programs developed via the SLOOP method.

Although the mapping to an existing executable language such as Smalltalk is straightforward, the purpose of this work was **not** to **maximise the efficiency of the executable program**. That is a topic for further research. Aspects that are currently implemented via reflection in metaclasses should form part of the **development environment**. A **SLOOP translator** could form part of such a development environment.

It must be noted that the choice of Smalltalk as the language to provide the SLOOP method with its object-oriented facilities was motivated to a large extent by the suitability of Smalltalk for experimental systems (this includes its reflective facilities). There is no reason why another object-oriented language could not replace Smalltalk in the SLOOP method when a fully fledged development environment is developed. However, such a language would have to provide at least the same capabilities as Smalltalk (except for the reflective facilities).

Desirable features of such a language would be proper support for **encapsulation** (in the case of C++ and Java this is somewhat illusory [ABV00]), **polymorphism** and **inheritance**. If further research shows that the SLOOP method should support **multiple inheritance**, then that would be a desirable feature of such a language as well. **Strong typing** would facilitate the detection of certain errors[5] during compilation and it would also be possible to implement compilation optimisations that could increase efficiency [Meye97]. A strongly typed language would then require the support for **genericity**, i.e. classes with formal generic parameters representing arbitrary types would have to be supported [Meye97].

As far as **software development tools** are concerned it would be very useful to have an **animated graphical trace facility**. Such a tool would be used during the design phase to generate animated execution traces. It would highlight an object on the screen whenever that object executes an unconditional parallel statement or whenever it executes a conditional parallel statement and the condition evaluates to true. The values of some of the instance variables as they are after the execution of the parallel statement could be shown. If sequential methods are invoked by the parallel statement, then the target objects could be highlighted in another colour. Such a tool would not replace the correctness reasoning described in Chapter 7. However, it could be used to aid **understandability**, since it would facilitate the **visualisation of event flows**.

Another aspect that could be investigated further is to find **more succinct** ways of presenting the informal correctness arguments, **without** going to the lengths of changing SLOOP into a formal method. The inclusion of **real time properties** into the formalism is another potential area for further investigation.

This research has produced very encouraging results. The SLOOP method facilitates solutions that are elegant, reusable, extendible, understandable and reliable. Further research would enhance the method, but it can already be applied successfully in its existing form. A solid foundation has been laid for creating high quality software systems.

[5] It would be possible to detect during compilation that a message is being sent to an object which does not implement that message.