Protocol Validation and Implementation: A Design Methodology Using LOTOS and ROOM

Neil Hart

Thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

under the auspices of the
Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario, Canada
August 1998

Acknowledgements

I would like to thank all the people who helped me in completing this thesis.

First, I would like to thank my supervisor, Professor Luigi Logrippo, for suggesting the topic and for his helpful comments.

I would also like to thank the members of the University of Ottawa LOTOS group for all their advice and assistance; in particular I would like to thank Daniel Amyot, Laurent Andriantsiferana, Brahim Ghribi and Jacques Sincennes. I am also grateful to Francis Bordeleau for his encouragement and comments.

I wish to thank Bran Selic, Neil Patterson and Ken Panton of ObjecTime for the training course they so generously provided.

Most of all, I should like to thank my wife, Alison Van Rooy, for all her support, encouragement and editorial comments!

Finally, I wish to express my gratitude to Motorola for their financial and technical support.

Abstract

Formal methods have been proposed as a means of expediting the creation of reliable software. The use of formal methods allows for clear and unequivocal specification of a system's design, and makes possible a form of prototyping that allows for formal validation against system requirements. However, the adoption of formal methods by industry has so far been slow.

It is proposed that one of the obstacles to the adoption of formal methods is the difficulty of bridging the gap between a formally-specified system and a working implementation. If this gap is too wide, the advantages of formal specification will be lost in the transition to implementation.

The methodology described in this thesis attempts to close this gap by demonstrating how a system may be described using LOTOS (Language Of Temporally-Ordered Specifications) and validated against requirements using two techniques: composition with agent scenarios and temporal logic model checking. The methodology then allows for the derivation of a model in the ROOM (Real-Time Object-Oriented Modelling) notation, which may be automatically converted to an implementation in the C++ programming language.

The methodology is illustrated with two small case studies. The first is the GPRS Tunnelling Protocol, used for transmitting protocol data units within the network of the General Packet Radio Service. The second study concerns authentication of users of the POP3 Internet mail protocol and demonstrates inheritance in LOTOS. Together, these case studies illustrate the salient points of the design methodology.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

As the infrastructure of our society comes to depend more and more upon computers, the production of reliable software becomes a correspondingly more important task. While we may be able to tolerate occasional failures in personal computer software, central systems (such as telephone switches) generally must be more reliable. Formal methods have been proposed as an aid in the creation of reliable software.

Hall points out that, while formal methods are not infallible, "...any system benefits generally from using at least some formal techniques."[1] Formal methods allow for unequivocal specification of system requirements, enabling developers and testers to verify that a finished system conforms to requirements. Furthermore, the use of formal methods may determine certain properties of a system at the specification stage, thus avoiding expensive error correction later in development. However, the adoption of formal methods in industry has not been extensive so far. As Vissers et al. point out: "At any case we can observe a situation which is far away from the large scale breakthrough of FMs."[2] Vissers et al. go on to point out that the important part of encouraging industry use of formal methods is its potential in supporting the whole implementation trajectory. The Lotosphere project (examined in some detail in section 2.2) attempted with some success to create just such a full design methodology, covering the whole development path from requirements to tested implementation.

[1]Hall [Hal90], page 12

[2]Vissers et al. [VvSP93], page 5. See also [CDH+96] for a discussion of the adoption of formal methods by industry.

1.2 Motivation

The motivation for this thesis draws from earlier work in the Telecommunications Software Engineering Research Group on the formalisation of the General Packet Radio Service. The industrial collaborator was interested in the possibility of deriving implementations, using the ObjecTime toolset, from LOTOS specifications. Also, as the ROOM notation does not include a formal validation semantics, there was interest in combining the notation with a formal description technique. Since the thesis was written, the ObjecTime toolset has been enhanced to allow for validation with respect to Message Sequence Charts. However, this validation may still not be as powerful as the validation techniques available with LOTOS that are explored here.

The primary goal is thus to demonstrate a design methodology which combines the formal expressive power of LOTOS with the ease of implementation of ROOM. LOTOS and temporal logic are used to express system requirements in a formal way. A LOTOS specification is then written, using an implementation-oriented style suitable for aiding the creation of an implementation using the ROOM notation. This specification is validated against the formal requirements to ensure that it behaves as expected. This validation stage can take advantage of all the tools that are available for LOTOS. In particular, temporal logic model checking requires state space expansion, for which a number of tools are available (see section 3.5.3). Finally, an implementation model is created using the ObjecTime toolset.

It is important to note that this thesis does not attempt to define a complete translation from LOTOS to ROOM. Rather, it defines a design, validation and implementation methodology that uses LOTOS for design and validation and ROOM for implementation. To this end, subsets of the LOTOS and ROOM notations are used. In spite of this limitation, a viable methodology has been obtained. Further, the thesis does not address the issue of automatic translation between the two notations. However, clear guidelines for manual translation for a user experienced in both notations are given. It is recognised that manual translation may introduce errors and so, to fully benefit from the validation of the LOTOS specification, automatic translation will ultimately be necessary. The development of automatic translation has been left for further research.

The thesis also does not attempt to provide a complete validation methodology, which would include 'high-yield' scenarios to reduce the required number of validation sequences. Instead, the validation sections in this document are largely illustrative, indicating how validation may form part of a design methodology.

1.3 Organisation of the Thesis

The remainder of this chapter explains the two main notations used in the thesis: LOTOS and ROOM. Chapter 2 surveys previous work in the field of implementation of LOTOS specifications, paying particular attention to the Lotosphere project. The following chapter outlines the design methodology proposed in this thesis, explains the validation methods used and the specification style advocated and details the method for deriving an ObjecTime implementation from the LOTOS specification. Chapter 3 concludes by illustrating the methodology using the simple alternating-bit protocol. The methodology is illustrated by a more substantive example in chapter 4, which describes the application of the methodology to the GPRS Tunnelling Protocol. A significant contribution of the thesis is a proposal for allowing inheritance in LOTOS specifications (see section 3.6.4). Chapter 5 illustrates how inheritance can aid in the development of the authorisation stage of a POP3 mail server by capturing the common behaviour between a basic authorisation technique and a more advanced one. The thesis concludes with an overview of the contributions made and a survey of possible future work.

1.4 Notations

The thesis presents a design methodology integrating formal methods for requirements capture and design, together with an object-oriented modelling technique for implementation. As a precursor to the discussion of the methodology in chapter 3, I will provide a brief overview of the two major notations used here; the formal description technique LOTOS and the object-oriented modelling notation ROOM.

1.4.1 The LOTOS Formal Description Technique

Introduction

The LOTOS (Language Of Temporally-Ordered Specifications) language was developed during the standardisation of the Open Systems Interconnection (OSI). It was recognised that such a complex set of protocols could not realistically be described using the informal natural language techniques typically used for system descriptions. In order for the protocol descriptions to be useful, they needed to be precise, yet implementation-independent. ISO[3] determined that formal description techniques (FDTs) would be best suited to meeting these requirements, and sponsored the development of LOTOS and ESTELLE. While ESTELLE was based on finite state

[3] International Standards Organisation

machine models, LOTOS was based on process-algebraic concepts derived from Milner's CCS[4] and Hoare's CSP[5]. The process-algebraic behaviour description notation was augmented with an Abstract Data Type (ADT) data description notation, Ehrig and Mahrs' ACT ONE [EM85]. After additional development, the language was finally standardised by ISO in 1989.[6]

Important features of the language

LOTOS has a number of important features which are relevant to the design methodology explained in chapter 3. Among these features are:

Synchronous symmetric communication Communication between processes is achieved through synchronous symmetric communication. This synchronisation can involve more than two parties, in which case they must all agree to synchronise simultaneously.

Atomic events All events are regarded as atomic and of zero duration.

Formal semantics A formal semantics is provided for LOTOS both in terms of inference rules, indicating the next state of the system, and in terms of expansion rules, which allow the semantics of all other operators to be expressed in terms of a small number of basic operators.

Executability The inference rules of the formal semantics of LOTOS make it possible to create execution environments which determine the description of the next state of the system given the current state and a chosen action. These environments allow for simulation (allowing a designer to observe a prototype of a system in operation) and validation (applying validation sequences describing required behaviour).

Basic operators

LOTOS specifications are composed of *behaviour expressions* that describe sequences of events. Events are referred to as *actions* and fall into two categories. Internal actions (denoted by `i`) can be executed by a process without synchronisation with other processes or the environment, while ordinary actions must synchronise and are

[4] Calculus of Communicating Systems, see [Mil89].

[5] Communicating Sequential Processes, see [Hoa85].

[6] For further explanation of the development of LOTOS and ESTELLE, together with the FDT SDL (developed within CCITT (The International Consultative Committee on Telegraphy and Telephony, now the ITU-T)), see Turner [Tur93]. For more detailed tutorials on LOTOS, the reader is referred to the papers by Logrippo et al. [LFHH92] and Bolognesi and Brinksma [BB87].

offered at synchronisation points called *gates*. The most basic behaviour expressions are those denoted by `stop`, which indicates that no further action can occur (ie. the system is deadlocked) and `exit`, which indicates that a process exits successfully. Actions can be sequenced using the action prefix operator, `;`, creating sequences such as `a; b; c; exit`. For the purposes of illustration, lower-case letters, such as `a`, `b` and `c`, will be used to denote actions, while upper-case letters, such as `B` and `C`, will be used to denote behaviour expressions. Thus, the general case of the action prefix operator is `a; B`, which means that the process synchronises on gate `a`, and then behaves like behaviour expression `B`, which can be a `stop`, an `exit` or other actions.

To indicate that alternative sequences of events are possible, LOTOS provides the choice operator, `[]`. This is placed between two or more behaviour expressions, and indicates that either (or any) of the specified behaviours is possible. For example, `a; B [] c; D` can either synchronise with gate `a`, then behave like `B`, or synchronise with gate `c`, then behave like `D`. Sometimes, the process with which this expression synchronises will force the choice; perhaps only synchronisation with `c` is possible. If the choice is not forced, LOTOS mandates non-determinism; either branch may be taken. Non-determinism can be made explicit in either of two ways:

1. Specifying the same initial action on either side of a choice operator indicates that the process can synchronise with that gate, but then behave differently. For example `a; B [] a; D` can synchronise with a process offering to synchronise on `a`, but the other process cannot determine whether the following behaviour expression will be `B` or `D`.

2. Using an internal action allows the process to go ahead and make a choice without synchronisation with another process or the environment. For example `a;(i; B [] i; D)` synchronises with gate `a`, but then makes an internal transition to a system that either behaves like `B` or like `D`.

Actions may be grouped together to form a *process*, which encapsulates its behaviour. The process definition takes the form:

```
process <process-name> [ <gate-list> ] :  <exit-behaviour> :=
    ...actions...
endproc
```

In this definition, `<gate-list>` lists the gates through which the process will synchronise. This list must include all the non-internal actions with the exception of those hidden by the `hide` operator (see below). The specification of `<exit-behaviour>` should either be `exit` to indicate that the process terminates by exiting, or `noexit` to indicate that the process either stops (using the `stop` action) or invokes another process. Recursive process definitions are allowed and, in fact, are the way in which repeated behaviour must be specified; LOTOS does not provide looping constructs. A further extension of the `<exit-behaviour>` allows for passing values to the next process.

Processes may be composed sequentially using the enable operator (`>>`). Given two processes, `P1` and `P2`, `P1 >> P2` indicates that if and when `P1` terminates with an exit, process `P2` will commence.

The *disable* operator, `[>` indicates that one process may interrupt another. For example, `P1 [> P2` indicates that process `P2` may interrupt `P1` at any point while `P1` is executing. The interrupt can only occur if the first action of process `P2` is enabled (that is, can execute). If the interrupt does occur, no further execution of `P1` happens, and execution continues with the actions of `P2`. If `P1` terminates unsuccessfully, `P2` may execute, while if `P1` terminates successfully (ie. with an `exit`), the actions of process `P2` are not available.

To enhance the encapsulation of LOTOS behaviour, the `hide` operator may be used to hide chosen gates from external synchronisation. For example, in the process definition below, only gate `g` is visible to other processes; gate `h` is only used for the internal processing of process `P1`.

```
process P1 [g] : exit :=
    hide h in
    (
        P1_1 [g, h]
    |[h]|
        P1_2 [g, h]
    )
endproc

process P1_1 [in, out] : exit :=
    ...
```

The above process definition also illustrates process instantiation. The process `P1_1` is defined with gates `in` and `out`. Specifying `P1_1` in the definition of `P1` indicates that the behaviour of `P1_1` will be inserted, with the gate `in` aliased to `g` and the gate `out` aliased to `h`. That is, wherever `P1_1` would have synchronised on `in`, it will now synchronise on `g`.

LOTOS processes may be composed in parallel in three different ways. The simplest way is through the *interleave* operator (|||), which indicates that the processes continue independently, save that they must both (or all, if there are more than two processes composed in parallel) agree to exit before the composition can exit. For example, in the LOTOS fragment below, both `P1` and `P2` must exit before process `P3` can be enabled.

`(P1[g] ||| P2[h]) >> P3[i]`

The second parallel composition operator, written as `||`, requires strict synchronisation on all non-hidden gates. In the fragment below, processes `P1` and `P2` must synchronise on gates `f`, `g` and `h` at every stage.

`P1[f, g, h] || P2[f, g, h]`

A related notion is the *generalised parallel operator*, written `|[gates]|`, which specifies a list of gates on which the processes must synchronise. The processes interleave on all other gates. Thus, in the fragment below, `P1` and `P2` must synchronise on gate `f`, but interleave on gates `g` and `h`.

`P1[f, g, h] |[f]| P2[f, g, h]`

LOTOS supports data specification through the abstract data type notation ACT ONE [EM85]. Although ACT ONE allows for the definition of many different data types, it is generally used here only for the definition of enumerated data types, as in the following example, defining two possible values of a message:

```
type message is
    sorts msg
    ok, err :-> msg
endtype
```

Given a definition of data types, LOTOS allows for value passing during synchronisation on a gate. An action can offer a value, written `! <value>`, or accept a value, written `? <variable>:<value-type>`. If two processes synchronise on a gate, and both offer a value, it must be the same value in order that the synchronisation can occur. If one offers a value and the other accepts a value, the value offered must be of the same type as that accepted by the other process, and the value is passed from one process to the other.

When values have been passed, we can impose conditions on further execution using the *guard* construct. For example, in the LOTOS fragment below, the value accepted at gate `g` must be `success` for the first branch to be taken, or `failure` for the second branch to be taken.

```
g ?  x:signal;
(
    [x = success] -> ...
[]
    [x = failure] -> ...
)
```

LOTOS provides a number of other constructs, and the issue of data specification is more complex than indicated here. However, the thesis relies on the subset of LOTOS syntax just covered.

Specification Styles

The issue of specification styles is debated in an important paper by Vissers et al. [VSvSB91], who distinguish between extensional and intensional definitions of systems. Extensional definitions are those which describe the system solely in terms of externally observable behaviour, while intensional definitions include mention of internal interactions and sequences of actions. The authors identify some of the conflicting requirements of specifications, pointing out that specifications should be written in an implementation-independent way, thus allowing implementors the maximum freedom possible. This requirement conflicts with another issue, however; the structure of the specification is likely to govern the structure of an implementation, so that a poorly-structured specification is likely to result in poor quality implementations. Thus, the authors argue that specification writers should avoid excessive implementation detail, while at the same time consider how the specification will be implemented. It should be noted that it is possible to transform LOTOS specifications, such that a specification written in a more abstract style can be transformed into a more implementation-oriented style. For more details, see the Lotosphere project (discussed in section 2.2).

In order to aid specification writers, Vissers et al. identify four major styles of specification. The first two styles, referred to as the monolithic and the constraint-oriented, are extensional in nature and are intended for the early stages of design, while the state-oriented and resource-oriented styles are more intensional, intended for later stages of design.

Monolithic style The monolithic style is characterised by the absence of hidden actions and the lack of parallel operators. The specification appears as a branching choice between sequences of actions. The prohibition of the parallel operators prevents the specifier from indicating possible functional decomposition of the system, making the specification implementation independent. However, the specification is likely to be longer and harder to read than those written in other styles.

Constraint-oriented style The constraint-oriented style relaxes the prohibition of parallel operators, producing more compact and readable specifications than those possible using the monolithic style. The entities composed using the parallel operators are not to be viewed as software entities, however, but rather as constraints on the possible sequences of actions executable by the system.

State-oriented style The state-oriented style makes explicit the global state space of a system. Because a specification in this style is written in terms of the global state space, it suffers from the limitation of hiding the distributed nature of a system. However, the style can be combined with the resource-oriented style described below to yield a composite style particularly well suited to the later stages of design. The resource-oriented style describes the distribution of a system's functionality over a set of resources, while using the state-oriented style within each resource provides information about the functioning of each resource. This state-oriented description of each resource may be used to develop the implementation of the resource.

Resource-oriented style The resource-oriented style represents a system as a composition of communicating resources. Observable, internal and hidden actions are all included, and the system is composed of resources synchronising at gates. This style is particularly well suited to implementation oriented specifications, as each resource represents a self-contained entity that may be implemented as such and then composed with other resources to create the system. Within each resource, any specification style may be used, including the iterative application of the resource-oriented style. By this means, we can support a sequence of functional decompositions by decomposing resources into component resources. At the lowest level, the functionality of a single resource should be simple enough that any of the preceding styles may be applied.

The specification style presented in this thesis uses the resource-oriented style to accommodate a functional decomposition of the system into a group of communicating actors. Within each actor, a modified form of the state-oriented style is used to describe actor behaviour. In the paper by Vissers et al., an explicit state variable is used, as in the LOTOS fragment below. The invocation of `QA_service1` on the third line starts the system in the `awaitQ` state. The definition of process `QA_service1` indicates the choice of states that the process can be in (`awaitQ`, `pendingQ`, ...) and the way that the process should behave according to the state it is in. A transition to another state is indicated by a recursive invocation, specifying the new state as the first parameter.

```
process QA_service[Q, A]:noexit:=
    ...initialisation...
    QA_service1[Q, A](awaitQ, q, a)
where
    process QA_service1[X, Y](s:state, x:question, y:answer):noexit:=
        [s = awaitQ] -> X?x1:question;
            QA_service1[X, Y](pendingQ, x1, y)
        [] [s = pendingQ] -> Y!x;
            QA_service1[X, Y](awaitA, x, y)
        [] [s = awaitA] -> Y?y1:answer;
```

```
            QA_service1[X, Y](pendingA, x, y1)
        [] [s = pendingA] -> X!y;
            QA_service1[X, Y](done, x, y)
        [] [s = done] -> stop
    endproc
endproc
```

The state-oriented style used in this thesis represents each state as a separate process. Thus, the example above might be rewritten as:

```
process QA_service[Q, A]:noexit:=
    ...initialisation...
    QA_service_awaitQ[Q, A](q, a)
where
    process QA_service_awaitQ[X, Y](x:question, y:answer):noexit:=
        X?x1:question;
        QA_service_pendingQ[X, Y](x1, y)
    endproc
    process QA_service_pendingQ[X, Y](x:question, y:answer):noexit:=
        Y!x;
        QA_service_awaitA[X, Y](x, y)
    endproc
    process QA_service_awaitA[X, Y](x:question, y:answer):noexit:=
        Y?y1:answer;
        QA_service_pendingA[X, Y](x, y1)
    endproc
    process QA_service_pendingA[X, Y](x:question, y:answer):noexit:=
        X!y;
        QA_service_done[X, Y](x, y)
    endproc
    process QA_service_done[X, Y](x:question, y:answer):noexit:=
        stop
    endproc
endproc
```

In the LOTOS fragment above, each state of the `QA_service` is represented by a different process. Any value parameters that are passed (in this example, `x:question` and `y:answer`) are extended state variables; there is no value parameter to indicate the current state of the system. The intention is that this style is better suited to hierarchical decomposition of states (see section 3.6.3), and better suited to implementation using the ROOM notation (see section 3.8).

Equivalence and Testing

The semantics of LOTOS may be understood in terms of a generalisation of finite state machine called a *labelled transition system*, or LTS. The LTS is often drawn as a tree, such that each of the nodes of the tree represents a possible state of the system, and the branches represent alternative execution sequences. The collection of all of the possible execution sequences is referred to as the *traces* of the LTS, and hence of the underlying LOTOS specification.

In order to check whether a LOTOS specification may execute a particular trace, the trace is composed in parallel with the specification and the combination is executed (usually by means of an execution tool such as LOLA (see discussion on page 21)). If the combination does not deadlock before the end of the trace sequence, then the specification can execute that trace. This notion of testing using traces is used by Brinksma to define a notion of *conformance* (see [BSS87], also [Bri89]). A specification B_1 is said to conform (indicated by **conf**) to B_2 if no deadlocks occur when testing all the traces of B_2 against B_1 that would not have occurred when testing against B_2.

The notion of conformance may be weaker than required, as it does not include robustness testing. That is, B_1 **conf** B_2 does not guarantee that B_1 does not exhibit traces that are not specified in B_2. Brinksma goes on to define the *reduction* relation, such that B_1 **red** B_2 if and only if B_1 **conf** B_2 and $Tr(B_1) \subseteq Tr(B_2)$, where $Tr(B)$ is the set of traces of specification B. Informally, if B_1 **red** B_2, then everything that B_1 does do is allowed according to B_2 and what B_1 refuses to do can be refused according to B_2. The reduction relation is then used as the basis of the notion of *testing equivalence*: if B_1 **red** B_2 and B_2 **red** B_1, B_1 and B_2 are said to be testing equivalent.

Brinksma goes on to define the extension relation (indicated by **ext**). We say that B_1 **ext** B_2 if B_1 does everything that B_2 does (though it may do more) and B_1 cannot refuse behaviour that B_2 does not refuse. The extension relation is used by Brinksma in a discussion of implementation: an implementation of a specification must do everything that the specification does, but it may do more. As with the reduction relation, extension may be used to define testing equivalence; if B_1 **ext** B_2 and B_2 **ext** B_1, B_1 and B_2 are said to be testing equivalent.

These notions of conformance, extension and testing equivalence will be seen to be important later, particularly when discussing inheritance in section 2.3.2.

1.4.2 Real-Time Object-Oriented Modelling

Introduction

The Real-Time Object-Oriented Modelling technique, also known as ROOM, was created to offer an abstraction method for developing real-time systems (see Selic et al. [SGW94]). The intention is that ROOM should allow developers to manage the complexity of modern systems through a number of support mechanisms. A key aspect of this complexity management is the use of graphical notations to represent both the structure and the behaviour of the system being designed. It is argued that the use of the graphical notation makes it easier to comprehend the operation of the system, thus reducing the possibility of design errors. ROOM specifies executable models, allowing designers to try out their designs before committing them to final code. Furthermore, the ROOM technique is supported by the ObjecTime toolset, which provides drawing tools for the graphical notations together with an execution run-time system to support simulation and debugging.

The term *real-time* is not necessarily clearly defined, and a number of different uses of the term are possible. Real-time in the ROOM notation describes systems which share a number of characteristics. Among these are:

Timeliness The system must respond in a timely manner, and late responses will have some deleterious effect.

Dynamic structure The system will typically execute over an extended period and its internal structure will change over time.

Reactiveness The sequence of events to which the system must respond cannot be determined beforehand; the system must be ready to respond to whatever occurs. Typically, the response of the system will depend on some internal state determined by the previous events, so real-time systems are regarded as state-dependent.

Concurrency Events may arrive from several points simultaneously, and serial handling is typically not sufficient; the system must have multiple concurrent threads of execution, each handling a different aspect of the system's behaviour or a different input event.

Distribution The system will typically not exist on a single node, but will involve distributed components.

While these items may not form a universal definition of real-time systems, the developers of the ROOM notation use them to define the subset of systems most amenable to treatment with the ROOM technique. In particular, the definitions may include systems not typically thought of as real-time, while excluding some systems (such as hard real-time[7]) that are usually included.

Because ROOM has been developed to manage complexity, it is natural that the notation embeds notions of object-orientation to allow abstraction and reuse. The ROOM practitioners regard objects as representing more than just Abstract Data Types (ADTs). Instead, objects are regarded as *logical machines*, active over extended periods of time, reacting to incoming events and exerting influence over their environment. Regarding objects as logical machines allows the integration of objects which may be implemented as hardware components, or which may be intangibles, such as a telephone call.

Object-orientation offers a number of valuable features in the decomposition of complex systems. In particular, a central concept of object-orientation is that of *encapsulation*, that requires objects to hide their internal structure and algorithms and to present a well-defined interface to other parts of the system. Making a defined interface available eases the process of using an object — the designer need only understand *what* the object does without having to worry about *how* that behaviour is achieved. Encapsulation also allows the internal structure or behaviour of an object to be modified, perhaps to use a more efficient algorithm, without concerning the designer about the effects of this change on other parts of the system.

Connected with the notion of encapsulation is that of *messages*. Using messages to communicate between objects whose internals are hidden by encapsulation allows us to define clearly the nature and contents of inter-object communication. Thus, when object A wishes to obtain some information from object B, it must send a request message and wait for a response. This messaging offers a cleaner way of programming than the approach of allowing A to access *B's* information directly, as may occur in ordinary functional programming.

A further advantage of the use of object-orientation is the use of inheritance to reuse design work. Objects are regarded as instantiations of *classes*, where classes describe the structure and behaviour of objects. Two classes which have some structure or behaviour in common may inherit that structure or behaviour from a *base class*. The base class need be written only once, removing the need for code duplication, and the derived classes need contain only behaviour that is new or different. Some object-oriented languages (such as C++) allow for multiple inheritance, in which a class can inherit from more than one base class. However, this feature can lead to ambiguous code (C++ has some notational conventions for resolving this ambiguity,

[7]Selic et al. describe hard real-time systems as those "where missing even a single deadline is considered unacceptable.", offering as examples "nuclear power stations, medical equipment, and aircraft control." [SGW94], page 21.

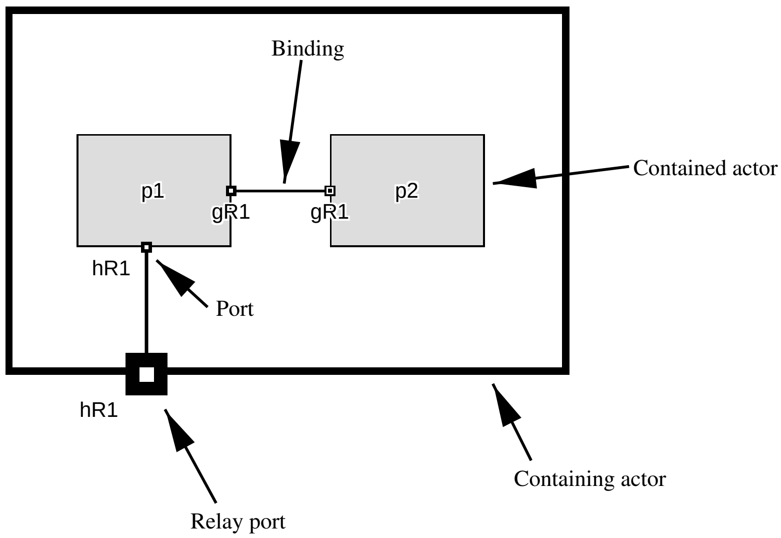


Figure 1.1: Representation of a hierarchically-structured actor in the ROOM notation.

though the code may remain confusing), and the designers of the ROOM methodology have chosen to not support multiple inheritance.

Actor Structure

ROOM models are based upon decomposition of a system into *actor classes*. Each actor class describes an object or group of objects with common structure and behaviour. Messages that may be passed between the actors are grouped into message sets, and these sets form the basis of protocol class definitions. The encapsulation boundaries of actors are broken only by *ports*, which allow messages to pass and are described by a protocol class. In this way, ROOM allows the designer to describe the precise set of messages that the actor can send and receive and preserve the notion of encapsulation. Actor class definitions include a description of behaviour that is regarded as orthogonal to actor structure. That is, the behaviour diagram should not be embedded within the structure diagram, but should be viewed separately.

The ROOM notation allows for the hierarchical structuring of actors. Actors may be decomposed into component actors, bound together by message passing. Component actors may communicate with the outside world through *relay ports* that connect a component actor's port to the encapsulation boundary of the containing actor. To model multiple instances of a system component, actors may be replicated, and actors communicating with replicated actors do so through a replicated port. See figure 1.1 for an example of the ROOM notation for an actor.

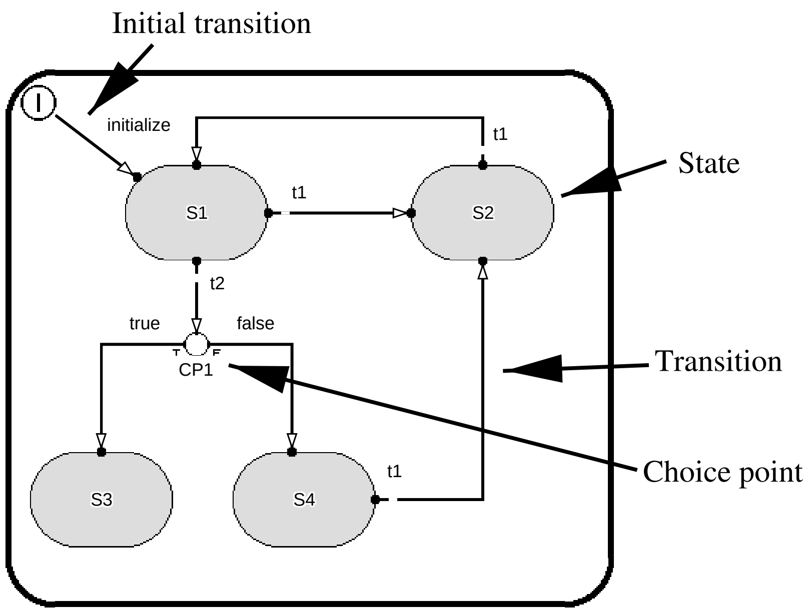


Figure 1.2: ROOMchart representation of the behaviour of an actor.

Actor Behaviour

Actor behaviour is defined in terms of *ROOMcharts*, a hierarchical state machine notation which owes much to Harel's *statecharts* [Har87]. ROOMcharts describe behaviour in terms of transitions between the actor's different states. Thus, the response of the actor to a given stimulus depends upon the state of the actor at the time. All transitions (with the exception of the initial transition, triggered automatically when the actor is created) must be associated with one or more trigger events, such as the reception of messages or a timer timeout.

Conditional behaviour may be specified in ROOMcharts either by associating guard conditions with transitions or through the mechanism of *choice points*. Guards are used to specify that a transition should be triggered if the triggering event occurs *and* a certain condition applies. The condition often involves testing the content of the triggering message and, if the condition is not satisfied, not undertaking the transition. Choice points are used when a given trigger should always result in a transition, but the transition may be to one of two destinations. The outgoing transition is associated with the desired triggering event and a condition is included in the specification of the choice point. If the condition is satisfied, one branch is taken; if not, the other branch is taken. See figure 1.2 for an example of the ROOMchart representation of the behaviour of an actor.

For the actor to influence its environment, actions may also be associated with transitions (so that if the transition is taken, the action is executed) or states (actions may be executed whenever the flow of control enters a particular state, or whenever it leaves a particular state). Actions may change internal variables or send messages to other actors, and this message passing can be asynchronous (through the `send` mechanism) or synchronous (through the `invoke` mechanism).

ROOMcharts inherit from statecharts the hierarchical state machine structure. This structure allows behaviour to be defined at a high-level of abstraction, and allows the states at this level to be decomposed into sub-states and thus provide greater behaviour detail. The hierarchical structure also allows states to be grouped together and group transitions (such as a common response to an *abort* signal) to be defined. The use of the hierarchy to define a common response to a signal can greatly improve the readability and comprehensibility of a state diagram. As well as supporting single *exit* transitions from a group of states, ROOM supports history transitions that return to the previous state.

Executable Code

The ROOM notation provides a formal, graphical notation to express both the structure and the behaviour of real-time systems. Because the notation is defined formally, it is possible to compile it into executable code, and this facility is provided by the ObjecTime toolset. The toolset provides graphical tools for the creation of ROOM models, together with editor windows for the specification of guards on transitions and action code. The models may be converted into corresponding C++ code that may then be compiled, using the standard system C++ compiler, to create executable implementations.

The ObjecTime toolset also provides debugging facilities through the Simulation RTS, allowing the designer to observe and control the execution of a model before compiling to the target platform. The debugging facilities include options to inject messages into ports, observe the traces of messages between actors, create Message Sequence Charts representing a single execution of the model, and so on.

Summary

While the ROOM notation may be used as part of a design methodology in its own right, it is used in this thesis largely because it provides a bridge between a formal model and a C++ implementation. By linking the LOTOS formal description technique with the ROOM modelling technique, the thesis provides a design trajectory from abstract specification to working C++ implementation.

Further details of the ROOM notation, together with diagrams illustrating its use, are to be found in the sections discussing its use in the design methodology (see section 3.8.1, for example). The following chapter surveys previous work related to the methodology outlined in this thesis. Chapter 3 outlines the methodology proposed by this thesis, indicating the steps required to produce a validated LOTOS specification and the means by which this specification can be used to derive a ROOM model.

Chapter 2

Previous Work

2.1 Introduction

In this chapter, previous work in the realm of applying the LOTOS formal description technique to the implementation of communications protocols will be considered. The principal work in this area is the Lotosphere project, a collaboration between a number of European universities and telecommunications companies. The Lotosphere project aimed to create an entire design methodology with tool support, using LOTOS as its major formal description technique.

Central to the Lotosphere project were the related notions of specification styles and transformations. Different specification styles allow LOTOS to be used to describe systems at varying levels of abstraction; some styles are better suited to the early stages of requirements capture and design, while others are better suited to later stages of design. However, if systems are to be described at different levels of abstraction, some means of moving from one level to another is needed, and the Lotosphere project focused on correctness-preserving transformations (see section 2.2) to perform this function. This discussion of transformations is followed, in section 2.2.2, by a review of work concerned with the process of deriving a C[1] implementation from a LOTOS description of a system.

Following the discussion of the Lotosphere project, other approaches to the implementation of LOTOS specifications are discussed in section 2.3.1. Research on possible relationships between object-orientation and LOTOS is discussed in section 2.3.2. The chapter concludes by examining a number of research projects which have applied LOTOS to the creation of industrially-applicable protocols (see section 2.3.3).

[1] For details of the C programming language, see Kernighan and Ritchie [KR88].

2.2 The Lotosphere Project

The principal aim of the Lotosphere project (see Bolognesi et al. [BvdLV95]) was to devise a design methodology that would offer enhanced productivity applicable to the industrial area of "large scale distributed telematic and information systems"[2]:

> The main objective of the ESPRIT II Lotosphere project (2304) was the development and industrial exploitation of a powerful, fully tool supported, system design methodology that is applicable to the entire system design and implementation trajectory.[3]

This industrial area is characterised by openness, a requirement that products produced by different manufacturers can inter-operate successfully. The intention of the Lotosphere participants was that it should be possible for a standards body to specify a protocol in an implementation-independent way (what they describe as part of the "public design culture"), and that there should exist a "private design culture", usable by implementors, that complements the public design culture. LOTOS is the proposed vehicle for these two complementary design cultures, and the Lotosphere project aimed to extend LOTOS from a specification language to a design language. This extension required the project to demonstrate how LOTOS could form part of a complete design trajectory, supported by tools.

The design methodology outlined in the Lotosphere project (see Quemada et al. [QAP95]) is based on conventional stepwise refinement (see Wirth [Wir71]). The first stage of design produces an abstract model of the system, that may be regarded as a black box description (see [QAP95], also Pressman [Pre87], chapter 13), concerned only with observable behaviour, but allowing systems to be related through the notion of testing equivalence (see Brinksma et al. [BSS87]). That is, if two systems are testing equivalent, they may also be described as black box equivalent, indicating that their observable behaviours are equivalent.

Following stages of the proposed design introduce more detail through successive refinements, marking the transition from black box descriptions to white box descriptions, revealing the internal structure of the system being designed. This addition of internal structure is also known as *functionality decomposition*, and consists of decomposing given functional units into smaller subunits. The subunits work in concert to fulfil the functional requirements of the encompassing unit. The final stage of such a series of refinements is a working implementation. This series of refinements integrates with a waterfall model of system development (see [Pre87], page 20).

It is obviously important that the process of functionality decomposition does not change the observable behaviour of the system, and it is here that the notion of testing

[2] Vissers et al. [VPvdL95], page 3.

[3] Vissers et al. [VPvdL95], page 15.

equivalence becomes important. The specifications produced during the refinements may be tested for equivalence with the preceding black box specification by using the LOTOS hide operator to hide the internal interactions. In this way, we can ensure that the behaviour of the new specification conforms to the behaviour of the previous specification. Further transformations may be needed to accommodate *functionality rearrangement*, in which the internal structure of a white box description is modified to accommodate architectural requirements, thus allowing easier mapping to system resources. These transformations may also be checked using the notion of testing equivalence.

Other possible transformations of a design include functionality reduction and extension. Functionality reduction is used to select between non-deterministic choices in a more abstract LOTOS specification, while extension adds new functionality. It should be noted that reduction and extension may not preserve testing equivalence.

The requirement that successive stages of a design be testing equivalent with each other may be satisfied in either of two ways. One approach is to manually transform the specification, and check the result in an *a posteriori* manner by testing for conformance with the preceding specification. Quemada et al. [QAP95] distinguish between validation, which they regard as correctness with respect to requirements, and verification, which is regarded as correctness with respect to a previous stage of the design. Both validation and verification are to be carried out using LOTOS testing, which involves composing a specification in parallel with a test case, and checking that the resulting composition does not deadlock.

Alternatively, one can use *correctness-preserving transformations*, a central part of the Lotosphere project. These transformations are ones which have been proven to preserve correctness. Using correctness-preserving transformations amounts to a formal proof that successive refinements of a specification are equivalent and should eliminate the need for a posteriori testing. One of the products of the Lotosphere project was a catalogue of correctness-preserving transformations (see Bolognesi [Bol92]), listing transformations which had been proven to retain equivalence.

In the Lotosphere project, a distinction was made between the early and late stages of design. In the early stages of design, the designer is concerned with establishing the architecture of a system. This architecture may include abstract functional elements as well as actual physical components. In the later stages of design, the designer is concerned with the production of an implementation-oriented LOTOS specification and from this, an implementation in a standard implementation language. These two parts of the late stages are sometimes known as the *implementation phase* and the *realisation phase*, respectively. The transformations carried out in the early stages of design include functionality rearrangement, reduction and extension, while the late stages of design are concerned with more implementation-oriented transformations, such as making states explicit (thus easing the creation of an extended finite state machine implementation).

The final stage of the Lotosphere design methodology involves the creation of a working implementation using the TOPO compiler (see Schot and Pires [SP95], page 70). Although some of the compilation process is carried out automatically from the LOTOS specification, the compiler also relies upon annotations concealed by the LOTOS comment operator. These annotations allow the designer to specify code in the C programming language to be executed at given points in the specification. The issue of creating C implementations from LOTOS specifications is covered in section 2.2.2.

2.2.1 Transformations of LOTOS

In order to create implementations of LOTOS specifications, it is frequently the case that an abstract specification must be transformed in some way in order to create a more implementation-oriented formal description. Van Eijk et al. describe the use of different specification styles in the implementation of a LOTOS specification of the sliding window protocol [vEKvS90]. The authors identify the four major styles of LOTOS specification as suited to different elements of a design trajectory. The constraint-oriented style is regarded as suited to the early stages of design, as it allows the specifier to describe constraints on behaviour as separate components which are composed together. The resource-oriented style offers a more implementation-oriented style, as each resource (which may map onto separate implementation environment processes or even separate hardware objects) is written as a separate part of the specification. For specifying the behaviour of each resource, the monolithic style may be used for relatively simple entities, while the state-oriented style is better suited for specifying more complex behaviour.

An essential aspect of the approach adopted by van Eijk et al. is that no parallelism existed within final protocol entities. In other words, each LOTOS process mapped onto a separate UNIX process, and there was no parallelism within a UNIX process. To ensure the absence of intra-process parallelism, the *parameterised expansion* was used (see the discussion on the next page). The result of this process was a specification consisting of a series of resources, each of which was specified in a state-oriented style. This specification was then hand-converted into C code, mapping LOTOS events to UNIX read or write actions.

Another transformational framework forms part of the LOLA tool (see Quemada et al. [QPF89], [QPF90]). LOLA was designed originally for labelled transition system verification, but the appearance of state space explosion led the research group to explore state space reduction techniques through compacting and parameterisation. One technique is *expansion*, which removes operators such as parallelism (`|[gates]|`), enable (`>>`) and disable (`[>`) by the application of the expansion theorems, which indicate the corresponding behaviour expressions in terms of action prefix (`;`) and choice (`[]`). To prevent expansion creating infinite specifications, LOLA recognises repeated behaviours and replaces them with process instantiations. A variation on ordinary expansion is *parameterised expansion*[4], which uses symbolic data values to reduce the size of the expansion. A parameterised expansion will grow in relation only to the basic behaviour, and not in relation to the product of the states of the behaviour and the number of possible values of the data components of the specification. Note that neither ordinary nor parameterised expansion can be applied to all LOTOS expressions. In particular, expressions that allow an unbounded number of processes to exist simultaneously cannot be expanded in any constructive fashion.

2.2.2 Implementing LOTOS in C

An important part of the Lotosphere project was support for the automatic, or semi-automatic, generation of C code from LOTOS specifications. An early example was the LOTOS Implementation Workbench (LIW) described in a paper by Mañas et al. (see [MdMvT89], also Mañas and de Miguel [MdM89]). The LIW system relied on a series of annotations to the LOTOS specification, allowing the implementor to indicate the C code which should be associated with each event. These annotations were written using the LOTOS comment construct so that they did not affect the compilation and simulation of the specification. Mañas's paper discusses many of the issues still relevant to the implementation of formal specifications, in particular arguing that full automation is not really plausible [MdMvT89]. LOTOS is sufficiently abstract that it specifies a number of different possible implementations, and so requires input from the implementor. Also, the use of abstract data types means that automatic implementation would be inefficient. For example, an ADT specification of integers would be typically unbounded; the data type would support arbitrarily large values. The specification may need only a restricted range, such as that handled by the C built-in type `int`, but the implementor must specify that this range is sufficient. Finally, LOTOS does not provide much, if any, support for the specification of hardware details, such as the particulars of communication channels and timing issues.

Because the LIW uses annotations, it is able to sidestep many of these objections to fully automatic code generation. For example, the implementor may associate each abstract data type with a C built-in type, indicating an efficient implementation. However, annotations are not without their problems. Most seriously, there is no way to check that annotations are faithful implementations of the LOTOS they are

[4]Parameterised expansion is also supported by the SELA tool (see Ashkar [Ash92]).

associated with. Also, while a LOTOS specification may indicate a non-deterministic choice, the corresponding implementation derived from compiling the annotations will generally be much more deterministic, and will likely have changed temporal fairness. Furthermore, annotating every event in a LOTOS specification runs the risk of creating an unreadable specification, thus negating some of the value of using formal, unambiguous methods for specifying complex systems.

These issues are taken up further in Mañas's 1993 paper in which the TOPO tool is discussed [MdMSA93]. In this paper, Mañas et al. introduce the idea of *environment fold-in* to produce clearer specifications. The protocol specification is composed in parallel with a process that represents the formal environment. The formal environment, in turn, acts as a bridge to the real environment and contains all the annotations necessary for creating real communication constructs, such as reading from and writing to UNIX sockets.

Wiedmer describes some of the experiences of the Lotosphere researchers in applying their design methodology to industrial systems [Wie95]. Two of the projects considered were a 'Mini-Mail' service for ISDN and the Transaction Processing application layer standard of OSI. The Mini-Mail application was implemented through two routes: one used C-Extended, an extension of the C language to include constructs for synchronous intertask communication, while the other used TOPO. The Transaction Processing system was created by writing a LOTOS specification in a style combining resource-oriented and state-oriented styles and then using TOPO to compile into C. The observations of the researchers included the impression that, while LOTOS was well-suited to producing structured specifications of large systems, data type specification using ACT-ONE was laborious and did not support the full range of data types required.

2.3 Other Related Work

2.3.1 Other Approaches to Implementing LOTOS

Karjoth's 1993 paper discusses a LOTOS toolset called LOEWE that supports a number of activities, such as editing, verifying and simulating LOTOS specifications [KBG93]. The toolset also includes two compilers for creating C implementations of specifications. The first compiler is based on direct code generation, rather than the creation of an intermediate representation, and the authors claim that this basis allows direct compilation of quite complex LOTOS constructs at the cost of more run-time support. The second compiler follows a more conventional technique, requiring the LOTOS specification to be transformed into a network of communicating finite-state machines that may then be converted relatively simply into implementation code.

Warkentyne and Dubuis's 1995 paper describes an intermediate approach that accommodates almost all LOTOS constructs and implements extended finite state machine (EFSM) specifications [WD95]. Their strategy supports value matching and multi-way rendezvous, but does not accommodate selection predicates. The COLOS compiler takes an annotated specification and uses the method outlined in Dubuis's 1990 paper to create a system in terms of automata and ports [Dub90]. The production of automata ensures that all behaviours are encapsulated in process instances, each of which will be executed by a single kernel thread in the final implementation. The implementation relies on run-time support for the ports that provide synchronisation facilities for inter-process communication. As with the LIW and TOPO tools, abstract data types must be annotated to indicate the corresponding C built-in type which is used in the implementation. The COLOS run-time provides a number of built-in gates, called *environment ports*, used for providing implementation services to the LOTOS system. In particular, timeouts (often represented in LOTOS as an internal action `i`) must be written in terms of interactions on the gate `clock`. Other environment ports provide keyboard, screen and socket services.

A recent paper by Yasumoto et al. describes a system for compiling LOTOS expressions into multi-threaded object code [YHA+96]. The authors claim the system supports almost all LOTOS constructs, although, as Warkentyne and Dubuis indicate, doing so involves substantial run-time support to handle some of the more esoteric parts of the LOTOS syntax. The researchers have created a portable thread library (PTL) which allows their run-time to be implemented on a number of platforms (at the time the paper was written these were restricted to UNIX varieties).

In the Yasumoto et al. study, the LOTOS specification is mapped to a series of threads, such that each action-prefixed sequence corresponds to a single thread. A shared data area is then used for interactions between these threads to indicate when threads should start, stop or destroy themselves. For example, the behaviour expression consisting of one expression enabling another (eg. `A >> B`) would be compiled as two threads, one each for expressions `A` and `B`, and a shared data area which process `A` would write to when it completes. Process `B` would not commence until it sees that process `A` has completed. On the other hand, a choice operator (eg. `A [] B`) would result in two threads, but once one thread commenced, the other would destroy itself. The team have also developed an intermediate formalism (called ASL/F) used for compiling ACT ONE ADTs into C code.

Yasumoto et al. have evaluated the performance of the code produced by their tool with code produced by both the COLOS compiler and the TOPO compiler. They point out that their tool and COLOS both retain fairness when compiling an alternative execution expression (such as `A [] B [] C [] ...`), while TOPO always selects the first expression. According to their tables of results, their tool produces code which executes at least twice as quickly as that produced by COLOS, and around 10–20 times as quickly as that produced by TOPO. However, this performance advantage over COLOS is not maintained in the presence of constraints (modelled in LOTOS using the parallel operator, ||). Furthermore, rendezvous are modelled using shared memory, so the applicability of Yasumoto's tool to the implementation of distributed systems is limited.

Amyot ([Amy93]) describes work on an implementation mapping LOTOS to the Occam language (a language used by the Inmos transputer). He points out that Occam, being based on CSP, has constructs such as two-way synchronisation which simplify the mapping from LOTOS. However, the version of Occam available in 1993 had very few data types and no support for data structures, limiting the options for mapping from LOTOS ADTs. Amyot describes a number of small case studies, such as a full adder, and demonstrates how they may be specified and validated in LOTOS, then implemented in Occam. Amyot concludes that Occam has drawbacks that, if overcome, would make it an attractive implementation language for LOTOS specifications.

It should be noted that most LOTOS operators translate into code in a relatively straightforward fashion, with the exception of parallel composition. The way in which parallel composition is defined in LOTOS assumes central concurrency control, which does not exist in the normal case of distributed implementation. Each of the implementation approaches described above has involved some solution to the problem of implementing parallel composition. For example, Yasumoto's group used a shared data area and considerable run-time support to cater for interactions between parallel processes. Warkentyne and Dubuis implemented LOTOS specifications in terms of communicating finite state machines, as did Karjoth. A similar approach is used in this thesis, although the methodology described here does not allow for multi-way rendezvous (see section 3.3).

2.3.2 Object-Orientation and LOTOS

Object-oriented analysis and design (OOA/D) has evolved over the last twenty-five years to occupy a central position in modern software engineering. Based on three fundamental principles — encapsulation, inheritance and polymorphism — object-orientation (OO) offers the facility of decomposing the work needed to develop a system, together with greater code robustness and maintainability than conventional structured programming techniques. However, LOTOS was developed before object-orientation had reached its current ascendancy, and so the formal description technique does not include explicit support for object-orientation.

Wegner distinguishes between three levels of adoption of object-oriented concepts in a language [Weg86]. *Object-based* languages support objects; program elements whose internal structures are *encapsulated*. *Class-based* languages additionally support classes, which represent the common characteristics of a collection of objects (the relationship between classes and objects is often described as *instantiation*, where an object instantiates a class). *Object-oriented* languages additionally support inheritance, whereby one class can inherit some of its characteristics from a parent class. Rudkin [Rud92] and Cusack et al. [CRS90] describe how the structure of LOTOS processes, which can interact only with other processes through defined gates, supports encapsulation. Furthermore, the instantiation of LOTOS processes with parameters may be regarded as support for classes. Thus, LOTOS may be regarded as both object-based and class-based. However, as we shall see, LOTOS does not adequately support inheritance, and so cannot be regarded as object-oriented.

Before considering how LOTOS fails to support inheritance in its present form, it is important to clarify what is meant by the term 'inheritance'. Some authors have described inheritance as a generalisation/specialisation hierarchy, allowing specialisations in which inherited features are "added, modified, or even hidden"[5]. Others have been more strict in limiting the way in which inherited features can be treated: "A feature should never be overridden so that it is inconsistent with the signature or semantics of the original inherited feature."[6] In the following discussion, I will follow Rudkin in replacing the loose term 'inheritance' by the term *subtyping*. A type s is a subtype of a type t only if 's satisfactorily substitutes for t in an environment expecting t'. It should be noted however that important object-oriented languages such as C++ also support specialisation in which inherited features are modified or hidden.

Mayr's 1989 paper [May89] links subtyping with the extension (`ext`) relation as seen in Brinksma [BSS87]. That is, an object type T', represented by a behaviour expression P', is a sub-type of object type T, represented by behaviour expression P, if any event sequence which is allowed by P is also allowed by P', and any event sequence refused by P' can also be refused by P.[7] Given a mapping between object operations and guarded choice in LOTOS, subtyping (or extension) corresponds to adding extra guarded choices. However, when extra choices are added, it important that non-determinism is not also added. That is, given a process skeleton:

[5] See Booch [Boo94], page 61.

[6] See Rumbaugh et al. [RBP+91], page 42.

[7] This relationship is formalised in Cusack and Lai [CL91] as: A process Q *extends* P if and only if: a) Q conforms to P, and b) traces(P) $\subseteq$ traces(Q).

```
process P1[gates]:noexit:=
    a1; a2; P1
    []
    b1; b2; P1
endproc
```

P1 can be extended by the sequence `[]c1;c2;P1`, but not by the sequence `[]b1; b3;P1`. The process `P1` already includes a sequence whose initial action is `b1`, so adding the new behaviour `[]b1;b3;P1` would add non-determinism. The extended `P1` would be able to synchronise with `b1`, and then non-deterministically either offer `b2;P1` or `b3;P1`.

As it stands, LOTOS does not provide support for subtyping as presented by Mayr, and he suggests an extension to LOTOS syntax that could be expanded by a precompiler to yield standard LOTOS. His proposed extension would define subtyping using the syntax:

```
process enhanced_printer1 is printer1[pr](id:source_id, s:state)
  with
  [s == ready] -> pr!line_feed!id; enhanced_printer1[pr](id, s)
```

This new syntax indicates that the process `enhanced_printer1` has the same gate list and parameter list as `printer1`, and the same behaviour but including the additional choice of the specified line.

Mayr's suggested extension to LOTOS is tentative and, in my view, not fully developed. In section 3.6.4 I explain how Mayr's suggestion may be developed to form a more powerful expression of inheritance in LOTOS.

Perhaps the most important paper discussing the relationship between object-orientation and LOTOS is that by Rudkin [Rud92]. Rudkin splits his discussion of inheritance into two parts. First, he establishes the way in which subtyping may be expressed in LOTOS, and the conditions subtyping imposes on the processes involved in an inheritance hierarchy. Second, he suggests a new LOTOS primitive process, `self`, which will support recursive process invocations in subtyped processes.

Rudkin indicates that subtyping is guaranteed through the extension relation, so that a process Q may be regarded as a subtype of P if Q *extends* P. Given a behaviour expression t, Rudkin then indicates that a behaviour expression $s = t[]m$ is a subtype of t if, among other conditions:

1. m is a behaviour expression not including any data type definitions;

2. m is stable, i.e. m does not have any initial internal actions;

3. The initial events of m are distinct from the initial events of t.

Having discussed inheritance in terminating processes, Rudkin goes on to discuss the issue of inheritance in recursive processes and introduces the new primitive process `self` in support. Recursive instantiations of a process that may be subtyped are written in terms of `self`, defined such that invocations of `self` are redirected to the invoking process.

While Rudkin's suggestion may have allowed full support for object-oriented specifications in LOTOS, his ideas were not adopted in the recent E-LOTOS standardisation, and so LOTOS remains, at best, a class-based language. However, the discussion in section 3.6.4 extends the ideas of Mayr and Rudkin to suggest how LOTOS may support a form of inheritance without necessarily requiring changes to the language standard.

While Mayr and Rudkins' papers were concerned with the theory of using LOTOS to support object-orientation, Hedlund describes an industrial application of LOTOS in an object-oriented development methodology [Hed93]. Ascom Tech AG had been using the Objectory methodology, and Hedlund describes the issues involved in integrating LOTOS into this methodology. The issues fall into two groups: *methodological*, or those determining how LOTOS could be integrated within Objectory, and *representational*, determining how to represent Objectory constructs in LOTOS. From a methodological point of view, an important aspect of the integration of formal methods into software engineering processes is that the formal method should not greatly disrupt existing processes. With this in mind, Hedlund suggests that LOTOS should be used to verify descriptions of specified object behaviour using existing techniques, rather than require system designers to replace their existing object behaviour notations with LOTOS. Hedlund's work on the representational aspects of integrating LOTOS into Objectory focuses on the following notion: while LOTOS does not explicitly support object-orientation, the well-defined interface of LOTOS processes may be used to model the encapsulation required in object-orientation. This mapping lead Hedlund to relate object communication to process synchronisation. The research also indicates how inheritance can be modelled so that a derived class can add behaviour to that inherited from its base class.

Hedlund concludes that while it is possible to integrate LOTOS into an object-oriented development methodology, care must be taken in the representation of object-oriented constructs in LOTOS. In particular, some constructs do not lend themselves to mapping into LOTOS, while others may be mapped only with care to ensure that the particular situation is suitable. As with the work of Mayr and Rudkin, the representation of inheritance described in Hedlund's paper is applicable only to subtyping, and can not be applied to situations in which inheritance is actually specialisation.

2.3.3 Implementations of Industrial Protocols

Building on the research described in earlier sections, a number of groups have attempted to apply a LOTOS-based development strategy to create implementations of protocol systems. Typically, these attempts have used stepwise refinement to successively create a more implementation-oriented LOTOS specification, followed by semi-automatic code generation using the TOPO compiler (see section 2.2.2).

The earliest work considered here is described by A. Fernández et al. [FQVM88]. Fernández's group designed and implemented an embedded system called PRODAT for use as the gateway between a satellite communication system and terrestrial data networks. The process described in the paper involved the creation of LOTOS models at several levels of abstraction. The most abstract level described a number of communicating processes, and subsequent levels refined the description by detailing the internal behaviour of the top-level components. To ensure that these refinements were consistent with the initial design, the relationship of testing equivalence was used. Testing was carried out at both a component and integration level by specifying sequences of actions in LOTOS and composing these tests in parallel with either components or the entire system specification. The test suite included both acceptance and refusal tests; the former are intended to complete successfully, while the latter are intended to deadlock, indicating that the system has correctly refused an incorrect action sequence.

Having reached a specification that was relatively implementation-oriented, the expansion theorems[8] were used to create LOTOS expressions composed only of choice and action prefix operators that can be readily represented as finite state machines. These finite state machine expressions were then converted into a MODULA 2 implementation. The implementation also required the creation of a kernel, written in MODULA 2, that implemented LOTOS synchronisation. The group concluded that the project had been successful, and that the development was finished within schedule despite the lack of tool support for much of the work. However, Fernández's paper concludes that the data description component of LOTOS is much less readily comprehensible than the behaviour component, and that many of the errors detected during component level testing were related to problems in data description.

Fernández's group followed their work on the PRODAT system with further research on a satellite communications system called CODE[9] [FMVQ92]. This later work was able to make use of the increased number of LOTOS tools then available, and the paper is in part an examination of the evolution of support for LOTOS. Because the satellite communication system had to support rigorous performance de-

[8] The expansion theorems describe the equivalence between expressions written using other LOTOS operators and expressions written using only choice and action prefix operators.

[9] Cooperative Olympus Data Experiment

mands, the group also carried out performance analysis to ensure that the protocols were satisfactory. This analysis included models based on queueing theory, simulation using the SIMSCRIPT language, and analysis using an extension of LOTOS (LOTOS-TP, which enhances LOTOS with support for timed and probabilistic behaviour). Having analysed the protocols, the group then created a specification of the system's behaviour in LOTOS, and used the TOPO tool to perform syntactic and semantic checking. This implementation-independent specification was refined to add more behaviour detail, and then transformed to make it suitable for implementation on the intended target system, a network of transputers. This specification was implemented partly by hand-coding OCCAM and partly by automatic translation to C using the TOPO compiler. The group concluded that tool support for the LOTOS design process had greatly improved, even over the relatively short period since their work on PRODAT, and commented that: "LOTOS is able to produce efficient implementation oriented specifications that can be almost automatically implemented over the selected hardware architecture."[10]

Fernández's paper was followed by a number of papers published in 1993, among them Ernberg's work on an ISDN architecture [EHM93]. Ernberg's group specified the ISDN system in a constraint-oriented style, inspired by earlier work using LOTOS to specify ordinary telephony systems (see Faci et al. [FLS90]). Having created a specification, Ernberg's group then attempted to validate the specification to ensure that it conformed to their informal requirements for an ISDN exchange. The group identified three main approaches to validation: generating a complete state space, interactively simulating the specification, and applying test cases in an execution environment. State space generation was ruled out as impossible for a large practical specification, and so validation was carried out in the early stages using simulation on the Hippo tool (see Tretmans [Tre89]). Having achieved a reasonable level of confidence in the specification, the group then applied a series of test cases using the LOLA tool [QPF90]. The validated specification was implemented using the standard TOPO method, annotating the specification with implementation-specific information. Ernberg's group conclude that LOTOS allowed them to detect logical errors at an early stage in the design process and so avoided extra work during the creation of executable code. Furthermore, the creation of an abstract specification allows for implementation in other languages.

The fourth paper considered here is by Azcorra et al. [AVACV93]. They specified, tested and implemented the D-channel layer 3 signalling protocol of ISDN, as defined in the CCITT Recommendation Q.931. The implemented protocol formed part of a prototype Integrated Services Private Branch Exchange (ISPBX) developed within the CEE ESPRIT Program. Azcorra's development process was split into three main

[10] Fernández et al. [FMVQ92], page 189.

phases: formal design, assessment and implementation. The formal design was carried out using the state-oriented specification style, felt to be the most straightforward and efficient for specifying detailed protocol sequences. The design incorporates a hierarchical element, in which the call handling procedure is viewed first in terms of a number of phases, each decomposed into an Extended Finite State Machine (EFSM).

The assessment phase involved the creation of a test suite applied to the specification to determine whether the specification behaved as expected. The researchers determined that two forms of testing were necessary. The first testing strategy (identified as *component level* by Fernández above) tested the Q.931 protocol entity by causing it to interact with both the upper and lower level interfaces. The second testing strategy (integration level) involved the construction of specification that combined two Q.931 entities with a backbone network, and a test of the resulting system through the upper level interfaces only. The test suite for the LOTOS specification was then run through the TOPO compiler to obtain test cases suitable for application to the final implementation. Azcorra points out that the implementation must be tested because it is possible for the annotation and semi-automatic code generation processes to introduce errors not present in the validated LOTOS. The process of testing the implementation is greatly eased by the facility to simply implement the test sequences created during the validation of the LOTOS specification.

An innovation introduced in Azcorra's paper is the *channel-gate* implementation approach, in which LOTOS gates are modelled as communication channels in the target environment (such as UNIX). The communication channel is an octet stream that supports only the reading and writing of some number of octets. Azcorra's group conclude that the LOTOS-based methodology, supported by tools, allowed for a cleaner system specification, and that the facility to run tests against the specification saved later effort at the implementation level. However, the group points out that LOTOS development methodology was quite immature, and that they had to develop a number of techniques in parallel with the creation of the ISDN implementation. Furthermore, the group indicates that it is important to consider at least some implementation details during the creation of the specification to avoid the production of an unimplementable specification that will require backtracking after implementation has started.

The final paper considered in this section is that of León et al., describing the design and implementation of a gateway between DSS1 and SS7 ISDN signalling systems, as described in CCITT Recommendation Q.699 [LYS+93]. The work was carried out in the framework of the MEDAS[11] research project with the twin aims of developing methodological guidelines and generating a set of tools to support FDT-based design. The authors point out that although LOTOS's mathematical theory allows

[11]Advanced Methodology for Communication Systems Development

for formal verification, such verification is impossible for realistically-sized specifications, so that testing equivalence is the only plausible method available for system validation. They also argue that the Lotosphere methodology does not constitute a complete life-cycle model and that it must be enhanced, possibly within the framework of the spiral software development model (see Boehm [Boe88]). As with previously-discussed projects, León's group used LOLA to apply test cases to their specification and then used TOPO to create automatically C code. They point out that the use of TOPO results in an implementation in which approximately 90% of the lines of C code are automatically generated, but caution that TOPO produces inefficient implementations of abstract data types, and that hand written implementations of data types are needed to improve efficiency.

The five papers reviewed above identify some of the major issues in the use of LOTOS for the creation of implementations of protocols. While some of the problems identified by the authors have become less serious with time (for example, Fernández's first paper describes the lack of tool support in 1988, though excellent tools were later developed), others remain obstacles to the effective use of LOTOS. One particular problem with LOTOS (identified in [FQVM88] and [LYS+93]) concerns the use of abstract data types, which are difficult and error-prone to specify and lead to inefficient code if hand-written implementations are not available. Furthermore, validation of industrial protocols through state space exploration is still infeasible, requiring that designers constrain validation through the use of test sequences. These issues are discussed further in the following chapter, and in the concluding remarks about possible future research.

An issue that the papers do not discuss, however, is that of object-orientation. Object-orientation has become increasingly prevalent in the software industry, with its proponents claiming that the use of encapsulation, inheritance and polymorphism lead to more reliable software and more opportunities for reuse of previous work. The methodology described in the following chapters attempts to redress this deficiency, by linking LOTOS with an object-oriented notation (ROOM) and indicating how encapsulation and inheritance can be features of LOTOS specifications.

Chapter 3

Design Methodology

3.1 Overview

This chapter describes the design methodology undertaken in this thesis, explaining the route taken from system requirements to the final working implementation. The chapter commences with a discussion of the rationale for the methodology, followed by a description of the subsets of the ROOM and LOTOS notations used. Section 3.4 outlines the stages of development of a protocol specification and implementation. Subsequent sections discuss the major stages; the formalisation of requirements and the creation of the specification, first in outline and then in more detail. Section 3.7 returns to the issue of formal requirements, this time to validate the LOTOS specification against initial requirements. This section also includes a discussion of design verification, which ensures that the internal behaviour of the system conforms to the designer's expectations. To conclude the design trajectory, the creation of a ROOM model and its compilation into a working C++ implementation is examined and the chapter concludes with a short example of the application of the design methodology to the creation of an alternating bit protocol implementation. Following chapters will exhibit the application of the methodology to more substantive examples.

3.2 Outline of the Design Methodology

The methodology uses the LOTOS formal description technique for the formalisation of system requirements and the construction of a formal specification, together with the ROOM notation for the creation of a system implementation. Combining these two notations allows us to take advantage of their individual strengths to produce reliable implementations of communications protocols. Among the advantages of using the LOTOS FDT are:

1. Support for varying levels of abstraction is a feature, allowing critical structural aspects of a design to be checked before expending effort on finer points.
2. There is a large body of knowledge concerning validation of specifications using state space exploration, temporal logic model checking, and so on.
3. Standardisation of the FDT by ISO has encouraged the development of a large number of design and validation tools that can be used in conjunction with each other.

In its turn, the ROOM notation provides a powerful hierarchical state machine notation for describing system behaviour, support for hierarchical structuring of system components and support for object-oriented inheritance to allow reuse of design work. Perhaps the most important advantage, however, is that the ROOM notation is supported by the ObjecTime toolset, which provides automatic code generation for a working implementation.

This thesis contends that the combination of these techniques allows for the creation of communications protocol software that is validated against requirements, and designed using object-orientation and encapsulation to allow for easier future enhancement and maintenance.

3.3 The LOTOS and ROOM Notations

Previous sections (1.4.1 and 1.4.2) have discussed the LOTOS and ROOM notations in general terms. For the purposes of this thesis, however, only a subset of each will be considered. In particular, the use of LOTOS is constrained to reduce the use of abstract, non implementation-oriented constructs. Applying these restrictions reduces the flexibility and power of LOTOS. However, the creation of a style based on a subset of the features of LOTOS provides patterns that designers can use to create practical, implementable specifications. The work of Azcorra et al. [AVACV93] (see page 31 for more details) indicated that it is important to ensure that specifications are written with implementation in mind to avoid creating unimplementable specifications.

The systems specified and implemented in the case studies consist of actors that may be hierarchical in structure (that is, actors may contain other actors that are hidden from outside view). These actors communicate with each other over asynchronous links, though synchronous communication will also be discussed. The behaviour of the actors is described in terms of hierarchical state machines that may have state variables. In order to simplify the design somewhat, only enumerated data types will be considered. Further work may extend this research to use more varied and powerful data types. The designs considered do not include group transitions, often used to model interrupt behaviour and consideration of interrupts is also left for future exploration.

These restrictions mean that the implementation models encompass the following subset of the ROOM notation:

1. Data classes are always enumerated types. The use of enumerated types simplifies the associated LOTOS specification style.

2. Actors, hierarchical or otherwise, have end ports and relay ports, but not internal ports. The use of internal ports is associated with actors that both contain other actors and have behaviour of their own, which would complicate the LOTOS specification style used here.

3. Actor behaviour is described using hierarchical state machines, but without group transitions or history transitions. Group transitions and history transitions are generally used to model interrupt behaviour, which is difficult to represent in LOTOS (see section 3.6.6) and has been left for further work.

4. Transitions between states are made directly or through choice points. Direct transitions are used when a transition may or may not be made, depending on the value of an incoming message, while choice points are used when a transition must always be made, but the state reached differs according to the value of the incoming message.

5. Actor communication may be carried out synchronously or asynchronously (see sections 3.6.5 and 3.8.3).

The specifications of the systems use the following subset of the LOTOS formal description technique:

1. The specifications may include the action prefix operator, the choice operator, the hide operator and guard.

2. Parallel composition is allowed, with the restriction that synchronisation may only take place between two processes. This restriction is justified by the fact that multiway rendezvous is not practical in implementations of distributed systems and is not supported by the ROOM notation.

3. Data types are always enumerated. The use of enumerated types greatly simplifies data type specification in this thesis. Extension of the method to include other data types is left for further work.

4. The enable and disable operators are not used. The enable operator is not needed in the state-oriented style used, while the disable operator is generally associated with interrupt behaviour, which has been left for further work.

5. Process invocation is used, including recursive invocations, though the latter will be constrained to maintain their gate lists in the same order. This restriction on gate lists is necessary because, in the state-oriented style used, recursive process invocation represents a self-transition on a state. If the gate list were allowed to change order, the behaviour of the state would change in a manner which would not clearly map to the ROOM implementation.

6. The specifications are written in a modified state-oriented style, elaborated in section 3.6.3.

The next section outlines the stages of the creation of an implementation, followed by a discussion of the formalisation of requirements using agent views and temporal logic. The specification structure designed to express the design constraints is then discussed, focused on its ultimate implementation with the subset of the ROOM notation outlined above. Once designed, the specification is validated against requirements in a process outlined in section 3.7. Finally, section 3.8 discusses the mapping between the LOTOS specification and a ROOM model that will yield a C++ implementation.

3.4 Stages of the Design Methodology

The methodology follows the sequence of steps enumerated below.

1. Requirements Analysis and Formalisation. The first stage in designing a distributed system protocol involves requirements analysis. The function the protocol is to serve must be decided, as a first step towards determining its characteristics. For example, ensuring reliable transmission may be a concern, or security may demand that users be authenticated before allowing access to certain services. While system requirements may be expressed informally through prose descriptions, expressing requirements formally makes possible the validation of the system design against requirements. Agent views may be employed informally to clarify the nature of the system to be designed, but may also play a more formal role in system design. See section 3.5.2 for more details of agent views and their use in formal requirements.

   The second technique considered here for formal requirements analysis is temporal logic (see section 3.5.3). Temporal logic may be used to express the relationships between temporally-ordered events, such as cause-effect and stimulus-response relationships.

2. Consideration of Distribution. The distribution of the system must be considered, in order to divide different aspects of the functionality of the system. A communication protocol is used between two (or more) physically separate entities, and the functionality of the system must be divided between these entities.

3. Further System Decomposition. The division of the system described in the previous section is continued, such that individual physical entities themselves contain multiple software entities, or objects. This division into software entities is the basis of a design methodology that may be described as object-based or object-oriented (see section 2.3.2 for an explanation of the distinction between object-based and object-oriented). By breaking up the system, smaller, more readily comprehensible units can be produced. LOTOS can support this process through the encapsulation of behaviour. That is, the system may be designed at a relatively coarse level of detail, and then refined by decomposing LOTOS processes into smaller components (examined in greater detail in section 3.6.2). At each stage, we may apply validation and verification techniques (see below), to ensure that the specification continues to conform to the requirements formalised earlier. Validating the design throughout the process allows correction of errors before much work has been invested.

   The use of decomposition allows construction of quite abstract specifications during the early stages of the design; these specifications are useful for establishing the general shape of the system and for documentation purposes. As specifications of increasing detail are created and approach the level of implementation, they can retain sufficient abstraction to maintain the big picture.

4. Specification Validation. Once fully detailed, the specification may be validated against the formal requirements to ensure it behaves as intended. This validation process is explored in greater detail in section 3.7. The use of formal specification techniques also allows verification of the system by checking for the absence of unwanted behaviour such as deadlock and livelock.

5. Implementation. Finally, the validated specification is used to guide the creation of an implementation (see section 3.8). While the validation process improves confidence in the functioning of the system, further confidence may be gained by testing the implementation against test cases derived from the requirements and the specification.

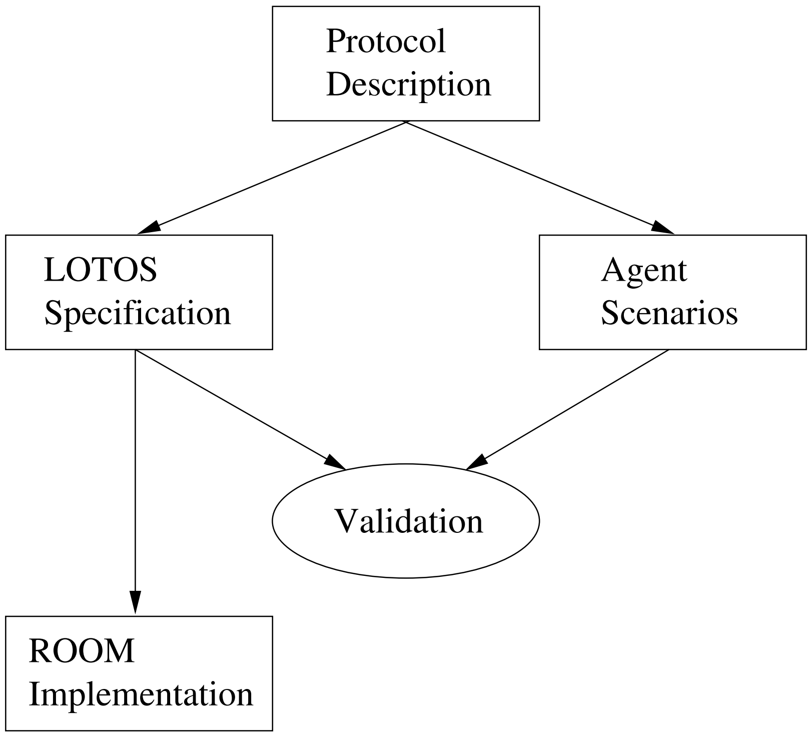


Figure 3.1: Outline diagram of the proposed methodology.

Figure 3.1 illustrates the broad outline of this methodology. The protocol description is used to create agent scenarios (step 1 above) and a LOTOS specification (steps 2 and 3). The specification is validated against the requirements expressed as agent scenarios (step 4) and, when the results of the validation are satisfactory, a ROOM implementation is derived from the LOTOS specification (step 5).

3.5 Formalisation of System Requirements

3.5.1 Correctness of Real-time Systems

The principal concerns in the examination of qualitative temporal properties are liveness, fairness and safety [Nis97].

Liveness is a temporal property that guarantees the eventual occurrence of some event. Another way to define liveness is the requirement to *make progress*. That is, a system exhibiting liveness will eventually reach certain important events, such as an output or a termination.

Fairness becomes important when a system may, at a given recurring state, non-deterministically choose between two or more transitions to new states. If one of the transitions is never chosen, the corresponding behaviour will never make progress. An example occurs in process scheduling, in which it is important that all processes make some progress before the system is considered truly responsive.

Safety refers to guarantees that certain events will never occur. Lamport explains this notion of safety as "something will *not* happen."[1]. It should be understood that the word 'safety' is not used here to refer to the conventional concept of safety as avoidance of accidents. Instead, safety is understood as a notion complementary to liveness. Liveness requires that the system make progress; safety requires that it not do so in a deleterious fashion. Real-time or reactive systems must combine both requirements: a system that does not exhibit safety may exhibit undesirable behaviours, while a system that does not exhibit liveness has no behaviour.

3.5.2 Agent Views and Agent Scenarios

One way to represent system requirements formally is to use *agent views*. Agent views were introduced by Clark and Moreira and offer a way of describing the behaviour of a system in terms of the interactions between the system and its environment or users (see Clark and Moreira [CM97a] and [CM97b]). Agent views are related to use cases, an increasingly popular means of describing system behaviour (see Jacobson [Jac92]). Both techniques are based upon describing sequences of operations or interactions that form part of a typical execution scenario. A collection of such scenarios allows for the construction of a behaviour tree, in which each branch represents an alternative path of execution. Agent views differ from use cases, however, in terms of the behaviour described. Clark and Moreira indicate that use cases include both interactions between the system and its users *and* internal operations required to support this external behaviour. Agent views, on the other hand, describe only the external interactions of the system, abstracting away from details of internal structure and operations. The term "agent view" refers to the whole behaviour tree of interactions — a single branch of the tree is called an *agent scenario*.

Clark and Moreira advocate the use of LOTOS to formalise the sequences of interactions, such that a single agent scenario can be represented by a single branch of the labelled transition system resulting from the expansion of the LOTOS specification of the agent view. Thus, the expected interactions of the system with an

[1] Lamport [Lam77], page 125

agent are expressed in terms of sequences of LOTOS actions. Furthermore, in the case in which more than one agent interacts with the system at once, the LOTOS processes describing each of their interactions with the system can be combined into a single process by means of LOTOS parallel operators. If two agents interact with the system independently, their corresponding processes are combined using the interleave operator (|||), while agents that must also synchronise with each other may be represented by processes that combine through the generalised parallel operator (`|[gates]|`).

Using LOTOS to formalise the notion of agent views provides a useful means for formalising system requirements. Because the agent views make no reference to internal structure, they can be constructed before real system design has begun. Moreover, this abstraction away from internals means that formalised agent views can form the basis of black box system validation (see section 3.7) at a later point in the design process. Individual agent scenarios describe sequences of expected interactions. If the LOTOS representation of an agent scenario is combined in parallel with the LOTOS specification of the system, simulation tools can be used to determine whether the desired sequences of actions can be observed. If the sequences do not occur, then the system does not meet the requirements expressed in terms of agent views.

3.5.3 Temporal Logic

Introduction to CTL

A second technique applicable to the formalisation of system requirements is temporal logic. Temporal logic is described as: "a special type of Modal Logic; it provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time."[2] Temporal logic extends ordinary propositional or predicate logic with temporal operators. A number of forms of temporal logic have been developed, and the one used in this thesis, *computational tree logic* (CTL), was proposed by Clarke et al. [CES86].

The formal syntax of CTL is as follows. AP is a set of atomic propositions.

1. Every atomic proposition $p \in \mathrm{AP}$ is a CTL formula.
2. If f_1 and f_2 are CTL formulae, then so are $\neg f_1$, $f_1 \wedge f_2$, AXf_2, EXf_1, $A[f_1 \mathcal{U} f_2]$ and $E[f_1 \mathcal{U} f_2]$.

The formal semantics of CTL is defined in terms of Kripke Structures, which are state transition graphs in which each state is labelled with the set of atomic propo-

[2] Emerson [Eme90], page 997

sitions which is true in that state. For a full description of the formal semantics, see Jonsson [JK90], page 183.

The semantics of CTL is understood in terms of discrete time branching trees of system states. The root of a tree represents the initial system state, described as the conjunction of all the propositions that are true in that state. Because time is regarded as discrete, subsequent states represent the state of the system in the next moments, again described as the conjunction of proposition truth values. Different possible future states of the system are represented by branching in the tree. The tree can be regarded as defining computation paths, each of which consists of a series of successive system states.

Intuitively, the symbols $\neg$ and $\wedge$ have their usual meanings of negating truth value and conjunction. X may be understood as a *nexttime* operator, and $\mathcal{U}$ as an *until* operator. A is used, in place of the usual $\forall$ symbol, to indicate *for all*, while E indicates *for at least one*. These operators may be combined to create temporal logic formulae that may be true or false of a given state within a computation tree. In the examples below, and illustrated in figure 3.2, the formulae are true of the top node in the corresponding graphs:

a. $AG(f)$ indicates that the formula f is true at all points from the first node onwards.

b. $EG(f)$ indicates that the formula f is true at all points from the first node onwards along *at least one* path.

c. $AX(f)$ indicates that the formula f is true at all the immediate successors of the first node.

d. $EX(f)$ indicates that the formula f is true at at least one of the immediate successors of the first node.

e. $AF(f)$ indicates that, along all paths leading from the first node, the formula f is true at some point in the future.

f. $EF(f)$ indicates that, along at least one path leading from the first node, the formula f is true at some point in the future.

g. $A[f1\mathcal{U}f2]$ indicates that, along all paths leading from the first node, the formula $f1$ is true until formula $f2$ becomes true.

h. $E[f1\,\mathcal{U}\,f2]$ indicates that, along at least one path leading from the first node, the formula $f1$ is true until formula $f2$ becomes true.

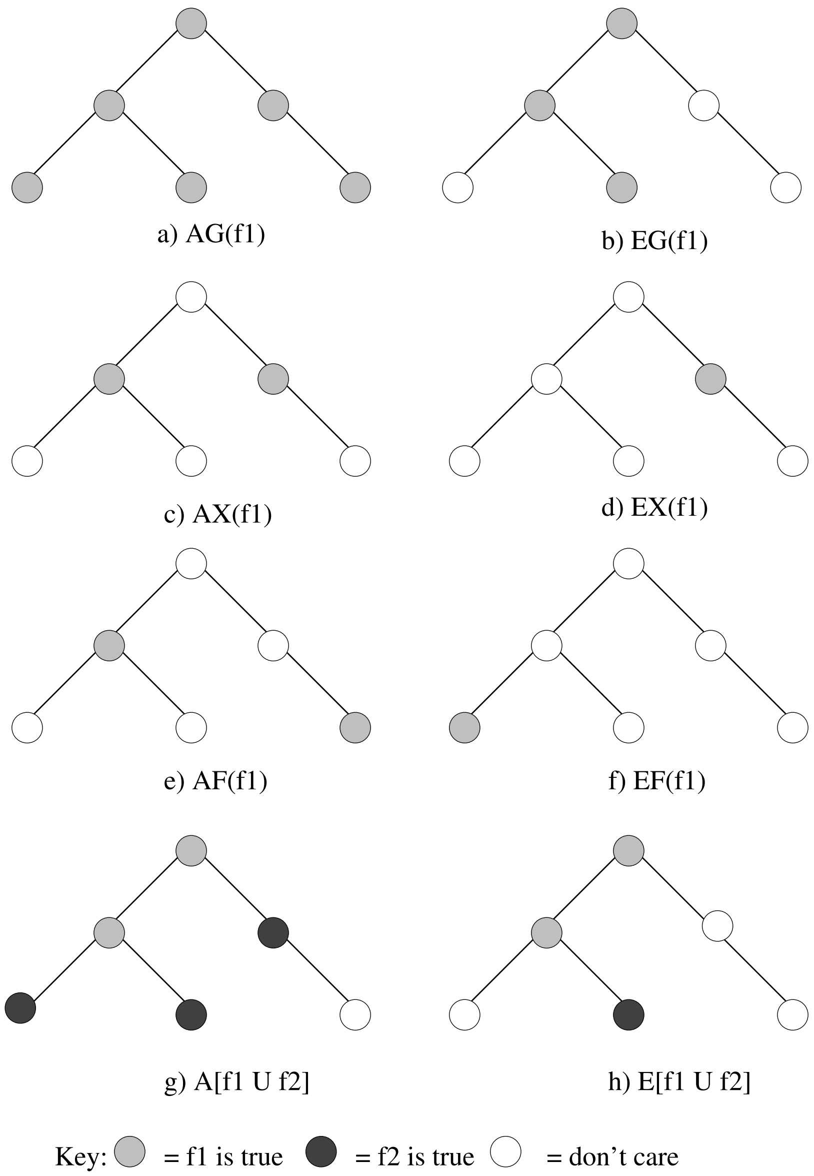


Figure 3.2: Graphs illustrating the meaning of various CTL formulae. Labelled nodes indicate that the specified proposition is true in that state.

In the context of LOTOS model checking, the atomic propositions of CTL are LOTOS actions, indicating that given actions occur. For example, $AF(a)$ indicates that synchronisation on gate a will occur at some point in the future along all execution paths. Thus, when CTL is used for the validation of LOTOS specifications, the Kripke structures that form the basis for CTL semantics differ from the usual Labelled Transition System model for LOTOS semantics. In Labelled Transition Systems (LTSs), actions are represented by transitions between nodes, while in the Kripke structures, actions correspond to the nodes themselves.

CTL allows formalisation of a system's intuitive requirements, indicating that certain properties are true of given states of the system. CTL may be used to express the fact that certain conditions must hold at some point in the future, or that one event must always precede another, for example. However, expressing requirements in terms of temporal logic requires some expertise. The need for this expertise has been contended as "...a substantial obstacle to the adoption of automated finite-state verification techniques..."[3] In order to overcome this obstacle Dwyer et al. have developed the notion of a *specification pattern system* [DAC98]. The following section describes the specification pattern system and its use in this thesis.

A Specification Pattern System

The development of the specification pattern system is inspired by the success of the use of patterns in other areas of software engineering[4]. Patterns capture the experience possessed by expert designers and make it available to others. Typically, patterns are specified in terms of a problem domain description, followed by a suggested solution. This presentation of the pattern makes it possible for designers to recognise similar problems posed by their own designs and apply the solutions described in the pattern documents.

Dwyer's group has extended the notion of patterns to the creation of temporal logic formulae describing desired properties of a system. The group has created a taxonomy of property specification patterns. For each pattern, an intuitive description of the property to be tested is given, followed by an expression of that property in a number of notations including linear temporal logic (LTL), CTL and graphical interval logic (GIL). For each property, a number of formulae are given, each describing a different temporal scope. The five scopes are:

Global indicating that the property holds over the entire program execution

Before indicating that the property holds up to a given state or event

[3] Dwyer et al. [DAC98]

[4] See Gamma et al. [GHJV94].

After indicating that the property holds after a given state or event

Between indicating that the property holds during any part of the program's execution from one given state or event to another state or event

After-until similar to the *between* scope, but including traces of the program's execution in which the first event or state occurs but the second never does.

The list of formulae is followed by an example of the application of the pattern, and the pattern concludes with an indication of the relationship between the pattern and other patterns in the taxonomy. Each pattern has been subject to peer review to check that the temporal logic formulae really do capture the informal requirement description at the head of the pattern. This peer review provides greater confidence in the correctness of the formulae than one would have in formulae one had written oneself and which had not been checked by others.

The taxonomy is split into three major branches: Occurrence patterns, Order patterns and Compound patterns. Occurrence patterns describe situations in which a given event should never occur (the Absence Property Pattern), must occur at least once (the Existence Property Pattern), should occur throughout a scope (the Universality Property Pattern) and so on. Order patterns describe the relationship between two events. For example, the Precedence Property Pattern describes the situation in which event P must always be preceded by event S. Compound patterns describe more complex combinations of events, such as those in which a group of events must be preceded by another group of events.

Some examples of patterns from Dwyer's taxonomy appear below.

1. The Absence Property Pattern describes a portion of a system's execution during which a given state or event never occurs. With *global* scope, this condition is expressed as:

$$AG(\neg P)$$

   That is, P is false in all states of the graph. With *before* R scope, the Absence Property Pattern is expressed as:

$$A(\neg P \,\mathcal{U}\, (R \vee AG(\neg R)))$$

   That is, along all paths, P is false until either R is true or until R will never be true. With *after* Q scope, the pattern is expressed as:

$$AG(Q \rightarrow AG(\neg P))$$

That is, for each state of the graph, if Q is true, then at that state and all following states, P is false. Finally, these two conditions may be combined to yield the *between Q and R* scope:

$$AG(Q \rightarrow A(\neg P\,\mathcal{U}\,(R \vee AG(\neg R))))$$

2. The Response Property Pattern (in the Order patterns branch) describes causal relationships in which an occurrence of the cause *must* be followed by an occurrence of the effect. With *global* scope, the requirement that P cause S is expressed as:

$$AG(P \rightarrow AF(S))$$

   That is, for all states of the graph, if P is true, then at some point in the future along all paths, S must be true. With *after Q* scope, the pattern is expressed as:

$$AG(Q \rightarrow AG(P \rightarrow AF(S)))$$

   That is, for each state of the graph, if Q is true, then at that state and all following states, if P is true, then along all paths at some point in the future, S must be true. An example of the use of the Response pattern is the requirement that a resource must be granted after it is requested. Specifying the *after Q* scope requires that another precondition (such as security authentication) be satisfied before the causal relationship between request and resource granting be applicable.

3. The Precedence Property Pattern (also in the Order patterns branch) describes the situation in which event P must always be preceded by event S. With *global* scope, this is expressed as:

$$A(\neg P\,\mathcal{U}\,(S \vee AG(\neg P)))$$

   That is, P must be false until S is true, or P must always be false. An example of the application of the Precedence Property Pattern is that a system resource must only be granted in response to a request.

By matching requirements with these specification patterns, an intuitive understanding of the precedence and causal relationships between events may be mapped into CTL formulae. These patterns have been used in the validation and verification of LOTOS specifications in this thesis (see, for example, section 4.6.2). The desired behaviour of a system was expressed in terms of intuitive requirements. These requirements were matched with the descriptions found in Dwyer's patterns, and the corresponding temporal logic formulae were instantiated with the events used in the specification. These instantiated formulae were then tested against the specification using the LMC model checker (see section 3.5.3). By this means, the specification could be validated against requirements without the need for error-prone derivation of temporal logic formulae from first principles.

Model Checking

Temporal logic formulae may be checked against formal specifications in two ways: general proof techniques and model checking. General proof techniques attempt to derive a formal proof of the validity of a formula. While powerful, this technique is infeasible for all but the simplest systems. For more practically-sized systems, model checking offers a way to prove that a given system satisfies temporal logic formulae. Model checking is based upon the generation of the full state space of a system in the form of a Kripke Structure, then the application of model checking algorithms to determine whether particular states in the tree satisfy the specified formulae. Because efficent algorithms exist (see [CES86]), this process can be automated and checked by a number of existing tools. The one used here is LMC, developed by Ghribi (see [Ghr92] and [GL93]). An important difference between LMC and other model checkers is that it was written specifically for LOTOS and converts the labelled transition system representing the execution of a LOTOS specification into a Kripke-like structure. Because the tool was designed for LOTOS, it supports the same syntax and so allows for the use of value-passing operators (`!` in LOTOS) and symbolic representation of data values.

3.6 Formal Specification

3.6.1 The Creation of a Specification From Requirements

As outlined in section 3.4, the creation of a formal specification from requirements involves the identification of discrete units within the desired system, each of which may be represented by a LOTOS process. The first stage in this decomposition of the system involves identifying the physical distribution of the system. A system may employ only two physical entities involved in client-server or peer-to-peer communication or, alternatively, may involve multiple entities, all of which may be regarded as peers.

Having identified the physical distribution of the system and the partitioning it imposes, we may then examine other aspects of the system requirements to determine whether further partitioning is necessary. For example, a client-server system will typically involve the requirement that a given physical server entity be able to serve multiple clients simultaneously. Particularly if these client-server relationships are not stateless (as would be the case, for example, if some form of security authentication precedes transactions), the server must maintain the state of each connection. This requirement may best be accommodated by constructing the server as a group of serving processes, one per currently active client. The server cannot be designed as a monolith — at least some form of decomposition is required. Further decomposition may be needed if separation of functional concerns is required. For example, we may wish to separate the part of the server responsible for authentication from the part responsible for transactions. By following this process, a sketch of the desired structure, composed of two or more entities communicating in order to provide the required functionality, is produced.

Because the thesis focuses on the practical implementation of formally-specified systems, the construction of the specification must concentrate on its ultimate suitability for implementation. The following sections explain some of the details of specification structure suitable for implementation by means of the ObjecTime toolset.

3.6.2 Structural Specification

The encapsulation that is such an important part of an object-oriented specification style is represented in LOTOS by using processes and the hide operator. Each process represents a single actor, which may contain other actors. To simplify the methodology, actors that contain other actors (henceforth referred to as *containing actors*) do not have behaviour of their own. That is, an actor may be either a containing actor, and thus have all of its behaviour determined by its component actors, or be an actor with behaviour. A containing actor will be represented in LOTOS in the following style:

```
process A[gate-list-A]:noexit:=
    hide hide-list-A in
    (
        A1[gate-list-A1]
            |[channel-gate-list-1]|
        A2[gate-list-A2]
        ...
            |[channel-gate-list-n-1]|
        An[gate-list-An]
    )
where
    process A1[gate-list-A1]:noexit:=
        ...
    endproc
    process A2[gate-list-A2]:noexit:=
        ...
    endproc
    ...
    process An[gate-list-An]:noexit:=
        ...
    endproc
endproc
```

The `gate-list-A` indicates the gates through which the actor will communicate with other actors. The use of the `hide` operator allows for communication between the constituent actors to be hidden and so provides for a means of encapsulation. The specification of processes `A1`, `A2` and so on within process `A` (using the `where` operator) hides those processes from parts of the specification outside process `A`. This hiding of structure contributes further to the encapsulation of the actors. Generally speaking, the gates listed in the channel gate lists (`channel-gate-list-1` and others) will also be listed in `hide-list-A`, as they represent the internal communication channels within actor A. Gates which do not appear in `hide-list-A` will be visible to actors outside A, and must be listed in `gate-list-A`.

Note that because the internals of actor A are hidden from view, top-level design can be completed without having to write the internal structure of actor A. The use of the `hide` operator to hide internal behaviour allows us to create a specification that may be compiled and validated at a very early stage of design. Also, this hierarchical structuring is not limited to the top-level of the design; one or more of the constituent actors of an actor may themselves contain actors with their own internal structure.

Each of the gates of the actor will be used to pass messages to and from other actors. As such, the messages that may be passed should be defined using the LOTOS abstract data type notation. A message type is defined as a LOTOS data type, and its possible values enumerated. Unfortunately, it is not at present possible to specify that a given gate will only pass messages of particular types. Instead, the designer must be careful to only use particular data types for communication through particular gates. For example, suppose that there is a gate `Ss` that should only pass the messages listed in the data type `Ss_out` below. LOTOS does not provide constructs to enforce this behaviour. Instead, the designer must ensure that the only messages that are sent on gate `Ss` are from the list of values indicated.

```
type Ss_out is
    sorts Ss_out
    ACTIVATE_request, DEACTIVATE_request :-> Ss_out
endtype
```

3.6.3 Behavioural Specification

With the structure of the system written, the behaviour of the component actors can be specified. The behaviour is written in a modified state-oriented style, using a process to represent each state. A process representing a state machine may be written in the following style:

```
process A1[gate-list-A1]:noexit:=
    initial-behaviour-A1
    A1-1[gate-list-A1-1](parameter-list-A1)
where
    process A1-1[gate-list-A1](parameter-list-A1):noexit:=
        ...
    endproc
    process A1-2[gate-list-A1](parameter-list-A1):noexit:=
        ...
    endproc
    ...
    process A1-n[gate-list-A1](parameter-list-A1):noexit:=
        ...
    endproc
endproc
```

The behaviour `initial-behaviour-A1` allows an actor to execute some initialisation behaviour before moving into its initial state, in this case `A1-1`. Each of the processes representing the states of the state machine (`A1-1`, `A1-2`, ..., `A1-n`) has the same gate list, and, if any state variables are required, they are contained within the (optional) parameter list, which is the same for each process. Again, the `hide` operator is used to hide the substates of state `A1`, so that the substates will not be visible to other parts of the specification.

Each of the states of the above state machine may itself be a state machine of the same format, yielding a hierarchical state machine. Alternatively, a state may not be subject to further decomposition, but instead be defined in terms of the transitions out of the state. For example, the following state `S1-1` has an outgoing transition to state `S1-2` triggered by the arrival of a message at gate `a`, and a transition to state `S1-3` triggered by a message at gate `b`:

```
process S1-1[gate-list-S1]:noexit:=
    a ?  msg:message_type;
    S1-2[gate-list-S1]
    []
    b ?  msg:message_type;
    S1-3[gate-list-S1]
endproc
```

More complex behaviour may be described, in which the state next reached is determined by the contents of a message, expressed in one of the formats below:

```
process S1-2[gate-list-S1]:noexit:=
    a ?  msg:message_type;
    (
        [msg = true] -> S1-4[gate-list-S1]
    []
        [msg = false] -> S1-5[gate-list-S1]
    )
endproc

process S1-3[gate-list-S1]:noexit:=
    a !  msg_value_1; S1-4[gate-list-S1]
    []
    a !  msg_value_2; S1-5[gate-list-S1]
    []
    a !  msg_value_3; S1-6[gate-list-S1]
    ...
```

The first format represents the situation in which the arrival of a given message will always result in a transition, and there are only two choices of next state. The second format is used when there are more than two choices of next state and/or when a given message may not always result in a transition.

By the application of the syntax illustrated in this section, it is possible to create quite complex state machine behaviour. Section 3.6.5 explains how the specification of state behaviour can be expanded to include sending messages to other actors.

3.6.4 Inheritance

One of the benefits offered by object-orientation is the facility of reuse, particularly through the mechanism of inheritance. If the behaviour of one system element can be regarded as an extension of the behaviour of another element, the first can be defined as a sub-class of the second, inheriting the behaviour they have in common. Using inheritance in this way allows designers to reuse code already written and tested, requiring new code only for the new behaviour. Unfortunately, as noted in the discussion of Rudkin's and Mayr's papers (in section 2.3.2), LOTOS does not provide direct support for inheritance. This section explains how limited support for inheritance can be provided without needing to change the LOTOS standard syntax.

Working with a slightly modified form of Rudkin's illustrative example, the following two definitions of a buffer process are considered:

```
process BUFFER [in, out](q:queue): noexit:=
    in?x:element; BUFFER[in, out](x appends q)
[]
    [q ne empty] -> out!hd(q); BUFFER[in, out](tl(q))
endproc

process BUFFER2 [in, out, flush](q:queue): noexit:=
    in?x:element; BUFFER2[in, out](x appends q)
[]
    [q ne empty] -> out!hd(q); BUFFER2[in, out](tl(q))
[]
    flush; BUFFER2[in, out, flush](empty)
endproc
```

The behaviour of `BUFFER2` adds an extra gate and an extra choice to the behaviour of `BUFFER`. Rudkin points out that a trivial inclusion of the `BUFFER` process within `BUFFER2` cannot be used because the recursive invocation of `BUFFER` would refer to the wrong process. His proposed solution is to add a new primitive process, `self`, which would provide the needed polymorphism. Although this addition provides a sophisticated solution to the problem, it requires changes in the LOTOS syntax and semantics, and is thus incompatible with existing LOTOS tools. The proposal made in the thesis is not as powerful nor as general as Rudkin's but works with the subset of LOTOS used here.

Following Mayr, a preprocessor would be used to process a specification written in a new syntax in order to yield a specification written in standard LOTOS. Mayr's syntax provided no facility for defining a derived process with a different gate parameter list or value parameter list from the base process. Changing the syntax slightly provides a more powerful form of inheritance, as seen in this example. Taking the specification of `BUFFER` above, we can specify `BUFFER2` as:

```
process BUFFER2 [in, out, flush](q:queue):  noexit:=
    extends BUFFER [in, out](q:queue):  noexit:=
    flush; BUFFER2[in, out, flush](empty)
endproc
```

The preprocessor can now replicate the code of `BUFFER`, *replacing the recursive instantiations of* ***`BUFFER`*** *with instantiations of* ***`BUFFER2`***, to give the same specification of `BUFFER2` as seen above. The new behaviour is appended to the old behaviour using a choice operator. This extension will work only given certain conditions on the two processes being specified. Process Q can only be defined as an extension of process P if the following conditions hold:

1. The gate parameter list of P must be a prefix (proper or not) of the gate parameter list of Q.

2. If process P has a value parameter list, it must be a prefix (proper or not) of the value parameter list of process Q.

3. If process P is defined as exiting, process Q must be defined as exiting, even if the added behaviour does not include an `exit`.

4. Appending the behaviour of Q to the behaviour of P must not introduce non-determinism. This requirement will be met if the initial actions of the behaviour of Q are different from the initial actions of the behaviour of P, and Q has no initial internal actions.

More generally, the operation of the preprocessor may be defined with reference to the following process definitions: a base process `B` and a derived process `D`. Note that each process is defined in terms of an `initial-behaviour` and a series of contained processes. The latter are optional; a process can be defined solely in terms of its initial behaviour. However, discussing inheritance in terms of this more general process structure offers two advantages over a more restricted structure. First, the discussion is more generally applicable than if it only applied to processes that did not contain other processes. Second, the state-oriented style described in this thesis requires that processes representing actors are structured with initial behaviour and contained processes representing states. As such, the discussion of inheritance is applicable to actors that inherit some behaviour from base actors, as seen in chapter 5.

```
process B[gates-B](parameters-B):exit-behaviour-B:=
    initial-behaviour-B
where
    process B1[gates-B1](parameters-B1):exit-behaviour-B1:=
        behaviour-B1
    endproc
    process B2[gates-B2](parameters-B2):exit-behaviour-B2:=
        behaviour-B2
    endproc
endproc

process D[gates-D](parameters-D):exit-behaviour-D:=
extends process B[gates-B](parameters-B):exit-behaviour-B:=
    initial-behaviour-D
where
    process D1[gates-D1](parameters-D1):exit-behaviour-D1:=
    extends process B1[gates-B1](parameters-B1):exit-behaviour-B1:=
        behaviour-D1
    endproc
endproc
```

The preprocessor must:

1. Ensure that `gates-B` is a prefix of `gates-D`;
2. Ensure that `parameters-B` is a prefix of `parameters-D`;
3. Ensure that if `exit-behaviour-B` is `exit`, so is `exit-behaviour-D`;
4. Apply the same conditions to the gate lists, parameter lists and exit behaviours of processes `B1` and `D1`;
5. Append `initial-behaviour-B` to `initial-behaviour-D` using a choice operator, ensuring that all references to `B1` are replaced by references to `D1`, and all references to `B` are replaced by references to `D`;
6. Append `behaviour-B1` to `behaviour-D1` using a choice operator, ensuring all references to `B1` or `B` are replaced as above;

7. Copy process `B2` into `D`, ensuring all references to `B1` or `B` are replaced as above;

8. Ensure that the addition of the new behaviour does not introduce non-determinism. Guaranteeing this is not possible in general, though some static checks may be applied.

Carrying out these actions will result in a process definition as below:

```
process D[gates-D](parameters-D):exit-behaviour-D:=
    initial-behaviour-D
    []
    initial-behaviour-B
where
    process D1[gates-D1](parameters-D1):exit-behaviour-D1:=
        behaviour-D1
        []
        behaviour-B1
    endproc
    process B2[gates-B2](parameters-B2):exit-behaviour-B2:=
        behaviour-B2
    endproc
endproc
```

This specification of inheritance differs from that presented in Rudkin [Rud92] in not requiring that a new primitive process, and differs from that presented in Mayr in allowing derived processes to have extended gate and value parameter lists.

3.6.5 Communication

The nature of communication between processes in the thesis' design will differ according to the physical distribution of the processes. Processes that execute on the same physical entity (typically those produced by functional decomposition of a higher-level process) may communicate synchronously. That is, the sender makes information available to the recipient and the exchange occurs instantaneously and completely reliably. Processes that execute on different physical entities will generally not be able to synchronise in this way. Instead, some form of message passing must be used, with its attendant risk of loss of information. Typically, one would not expect the sender process to block awaiting correct transmission, but instead to proceed and perhaps expect an acknowledgement at a later time.

Synchronous communication is the fundamental form of inter-process communication supported by LOTOS, as indicated in the example below:

```
(...; g ! message1; h ! data; ...)
|[g]|
(...; g ? message:message_type; ...)
```

In the above example, both processes synchronise on gate g, passing the value `message1` from the upper process to the lower process. If one process reaches this action before the other, it must wait, but once both processes are ready to synchronise, the communication is instantaneous and both processes proceed to the next action.

While synchronous communication is specified quite naturally in LOTOS, asynchronous communication must typically be specified using an intermediary process to represent the transmission medium. In the example below, process `P1` must synchronise with gate `in` of the data channel to transmit a message, but `P1` may then continue without waiting to see whether `P2` receives the message. The transmission completes when process `P2` synchronises with the gate `out` of the data channel. The specification fragment below uses the gate relabelling characteristic of LOTOS process invocation. The relabelling means that the invocation of `channel[a, b]` may be regarded as binding the gate names `a` and `b` to the gates `in` and `out` in process `channel`. Thus, the first action of the process `channel` will synchronise with gate `a` in process `P1`, while the second action will synchronise with gate `b` in process `P2`.

```
    P1[g,a] |[a]| channel[a,b] |[b]| P2[h,b]
where
process channel[in,out] : noexit :=
    in ? signal:signal_type ? payload:payload_type;
    out ! signal ! payload;
    channel[in,out]
endproc
```

Process `P1` can initiate a transmission using a send operation of the form:

```
a ! signal ! payload;
```

while process `P2` receives the data using an expression of the form:

```
b ? signal:signal_type ? payload:payload_type;
```

This basic channel specification models a channel that always transmits data correctly. Modifications are necessary to model a channel that may arbitrarily lose messages, using the LOTOS internal action `i` to model the non-determinism of channel data loss. Thus an unreliable channel specification might look like:

```
process channel[in, out] : noexit :=
    in ? signal:signal_type ? payload:payload_type;
    (
        out ! signal ! payload; channel[in, out]
    []
        i; channel[in, out]
    )
endproc
```

Using this channel specification may lead to problems when automated validation tools are used, however, as it is possible for the channel to continually lose messages, leading to an infinite state space expansion. If this loss is likely to be a problem, an alternative channel specification may be used, in which, for example, the channel can lose one message, but not two successive messages:

```
process channel[in, out] : noexit :=
    in ? signal:signal_type ? payload:payload_type;
    (
        out ! signal ! payload; channel[in, out]
    []
        i;
        in ? signal2:signal_type ? payload2:payload_type;
        out ! signal2 ! payload2;
        channel[in, out]
    )
endproc
```

These two alternative channel specifications may be appropriate in different circumstances. For example, in the GTP system described in chapter 4, the protocol allows only for a certain number of retransmissions before aborting and returning an error code. Thus, the GTP system can be specified using the first unreliable channel specification, which loses an arbitrary number of messages, and then the test sequences can be written to allow for either successful transmission or the error code. On the other hand, the ABP system described later in this chapter responds to continual message loss by continuing to retransmit the messages. In this case, use of the first channel specification would mean that verification tools would never terminate. The second channel specification, on the other hand, allows verification that the protocol can handle some message loss without the tests running for an excessively long period. As described in section 3.9.2, further modifications allow a bidirectional channel to be modelled.

3.6.6 Interrupts

Reactive systems may need to be ready, at any point, to react to an exceptional event. Events of this sort might include detection of critical environmental conditions (such as high temperatures or power failures) or top priority system events. In an embedded system, these might be handled by processor interrupts. Interrupts provide the facility for an event to be handled at any time. Typically, once interrupt processing has completed, the system continues where it left off. Alternatively, the system may exit or restart from a particular state.

This section will present two possible ways of representing interrupts in LOTOS. The first is of use when, after the interrupt handling code is complete, the actor should return to a specific state. The second option is for the actor to return to the state it was in when the interrupt occurred. Both methods suffer from some flaws, resulting from limitations in the standard LOTOS language. The section concludes by briefly reviewing an enhancement to LOTOS by Stepien, developed further by Hernalsteen and Février that adds a new operator to better support interrupt behaviour (see [Ste94] and [HF97]).

Specifying termination of a system, or restarting from a single state is relatively simple in LOTOS. The disable operator (`[>`) can break out of any behaviour expression or process. In the example below, whenever process `P2` is ready to synchronise on gate `d`, the expression `a; b; c; exit` will be disabled and the second action sequence (`d; e; exit`) will be executed:

```
((a; b; c; exit) ▷(d; e; exit))
|[d]|
P2[d]
```

To ensure that the system returns to a given state after processing the interrupt, the enable operator may be used to indicate that processing should restart. In the example below, process `Int` describes the behaviour to be executed when an interrupt is received, while `P1` indicates the normal processing of the system. The process `P1Int` behaves like `P1` until the interrupt triggering event (`d`) occurs, after which the behaviour of `Int` is executed to completion. The system then returns to behaving like `P1`, until another interrupt occurs, and so on.

```
process Int[d, e] : exit :=
    d; e; exit
endproc

process P1Int[a, b, c, d, e] : noexit :=
    P1[a, b, c] ▷(Int[d, e] ≫P1Int[a, b, c, d, e])
endproc
```

The above discussion of the disable operator may be applied to the implementation-oriented LOTOS style developed in this chapter as follows:

```
process P1[gate-list]:noexit :=
    IntS1[gate-list]
where
    process IntS1[gate-list] : noexit :=
        S1[gate-list]
        [>(P2[gate-list] >> IntS1[gate-list])
    endproc
    process S1 [gate-list] : noexit :=
        ...
    endproc
    ...
    process P2 [gate-list] : exit :=
        ...
    endproc
endproc
```

In this example, the actor represented by process `P1` starts in initial state `S1`. At any point, the behaviour of `P1` may be interrupted by `P2`. Once the interrupt has occurred, the behaviour of the actor will be that of process `P2` until the process exits. The enable operator (`>>`) indicates that when process `P2` exits, the actor will return to its initial behaviour through recursive invocation.

The presentation of interrupt behaviour above suffers from two main flaws. First, the actor must return to a specific state, regardless of the state it was in when the interrupt occurred. It is more usual in reactive systems to resume processing at the point at which the interrupt occurred. The second and more significant flaw is that the interrupt can occur at any point. Although individual LOTOS actions are defined as atomic, there is no provision in LOTOS to make a sequence of actions atomic, so it would be possible for an interrupt to occur part way through the processing representing a state transition, for example. Both of these flaws are resolved in the second means of specifying interrupt behaviour, taken from Hernalsteen and Février [HF97]. The process that may be interrupted is modified so that at any point at which an interrupt may occur, a `suspend; resume` sequence is inserted. These actions synchronise with the interrupting process to control the behaviour of the interruptible process. Thus, an actor may be specified as:

```
process P1 [gate-list] : noexit :=
    S1[gate-list, suspend, resume]
    |[suspend, resume]|
    P2[gate-list, suspend, resume]
where
    process S1 [gate-list, suspend, resume] : noexit :=
        <state-behaviour>
        []
        suspend; resume; <state-behaviour>
    endproc
    ...
    process P2 [gate-list, suspend, resume] : exit :=
        suspend;
        ...
        resume;
        P2[gate-list, suspend, resume]
    endproc
endproc
```

This actor starts in state `S1` and its behaviour evolves as specified by the state behaviour of each state. However, if an interrupt occurs as state `S1` is entered, process `P2` will synchronise on the `suspend` gate, and no further execution can occur within `S1` until the `resume` action occurs at the end of `P2`'s execution.

This means of specifying interrupt behaviour resolves the two issues identified above. Interrupts are limited to occurring in particular places, thus preserving the atomicity of transitions. After the interrupt handling has occurred, execution continues at the point at which the interrupt occurred. However, the readability of the specification is markedly reduced, particularly if the original behaviour of the states and transitions was quite complex. Hernalsteen and Février conclude that without an operator designed for the purpose, LOTOS "...is not able to represent correctly the suspend/resume concept which is essential for the design of real-time systems."[5]

Stepien suggests the addition of a suspend/resume operator that would allow one process to suspend another, resuming execution of the first process only when the second has completed [Ste94]. This suggestion was developed further in the paper by Hernalsteen and Février [HF97], and has since been adopted into the E-LOTOS standard. The operator is written as `[g>` and specifies that the process on the left of the operator may be suspended by the process on the right through synchronisation on the special action `g`. For example, `P1 [g> P2` indicates that process `P1` may be suspended at any point by process `P2`. If this suspension occurs, process `P2` is executed

[5] Hernalsteen and Février [HF97], page 404.

until it terminates, at which point the execution of `P1` continues. The suspension may occur repeatedly, modelling the behaviour of reactive systems in which interrupts may occur more than once.

Given the recognised difficulty of specifying interrupt behaviour using standard LOTOS, the methodology described here does not incorporate interrupts. Development of the methodology to include interrupts using the special constructs in E-LOTOS is left for further research.

3.7 Validation of the Specification

Validation of the specification may be performed in two ways. Validation against initial requirements may be regarded as *black box* validation (see section 2.2), because we are concerned only with observable behaviour. Validation that exploits the designer's knowledge of the full internal structure of the system may be regarded as *white box* validation. Between these two types of validation is *grey box* validation[6]. Grey box validation may be carried out if one assumes knowledge of the interactions of system entities without assuming knowledge of the internal behaviour of those entities. This latter form of validation may be useful to determine whether the correspondence between internal behaviour and observable behaviour of a protocol system is as expected.

Note that a distinction is often drawn between the terms 'validation' and 'verification'. The term 'validation' applies only to confirming that end-to-end behaviour requirements are met. The term 'verification' is used to describe any checking of a design that includes the internal behaviour of the system. In the sections that follow, 'black box validation' will refer to checking external behaviour, while 'grey box verification' will refer to checking against internal behaviour.

Two validation methods are used in this thesis.

1. Validation sequences are used to check requirements expressed in terms of desired sequences of observable events. The LOTOS specification of a system is composed in parallel with LOTOS sequences representing agent scenarios. The resulting compositions are then executed to ensure that the system can support the behaviour described by the scenarios. Because the sequences deal only with observable behaviour, they correspond to the black box validation mentioned above.

2. Temporal logic is used to confirm temporal properties of a system, such as that X causes Y. These properties may refer to observable events only, providing

[6]See Castanet and Koné [CK94] and Petrenko et al. [PYD95] for more details on grey box testing, validation and verification.

additional black box validation, or may refer to internal events as well, typically providing grey box verification. The latter is particularly useful to check the relationship between observable events and internal processing (see, for example, section 4.6.2).

3.7.1 Validation Using Agent Scenarios

An agent scenario is an expected sequence of events, seen from the point of view of one or more agents interacting with the system. Applied to communications protocols, the agents are regarded as those entities that request that messages be sent, and that receive messages at the other end. Thus the only visible events are message requests and receptions; events occurring on the link within the communication system are not visible and are not described in agent scenarios. Given the generalised communication protocol example below, the agent scenario must only be expressed in terms of events on the gates `s` and `t`, but not in terms of events on the hidden gate `l`:

```
process communication_link[s, t]:noexit:=
    hide l in
    (
        sender[s, l]
        |[l]|
        receiver[t, l]
    )
endproc
```

The expected sequence of events must be written in a format compatible with the system specification. A synchronisation on the special gate `success` is added as the last event, and the sequence is encapsulated within a process (see section 4.6.1 for an example). The LOLA tool is then used to check whether the sequence is observable. The tool composes the test process in parallel with the system specification and carries out a complete state space expansion (see Miguel et al. [dMAQM95] for details on the operation of LOLA) . The result of the expansion will be one of three results: may pass, must pass or reject. A *may pass* result indicates that, for at least one execution of the specification in parallel with the test sequence, the `success` event is reached. A *must pass* result indicates that for all executions of the specification in parallel with the test sequence, the `success` event is reached. A test for which the `success` event is never reached yields a *reject* result.

While section 3.5.2 described agent scenarios in terms of single execution paths, writing tests in this format does not allow for non-determinism in the specification. When specifying systems that communicate asynchronously, and which may lose messages, it is possible that a given scenario may evolve in a number of ways, depending on whether messages are lost. In order to accommodate this choice, the correct response of the system to message loss must be determined, and the test written including branches representing all of the possible behaviours. Thus, if the possible outcomes of sending a message along the link illustrated above are: the receiver reports that a message was received, or the sender reports that the transmission failed, a suitable test sequence might be written:

```
process communication_link_tester[s, t]:noexit:=
    s ! send_message;
    (
        t ! message_received;
        success;
        stop
    []
        s ! transmission_failed;
        success;
        stop
    )
endproc
```

Note that because agent scenarios are written only in terms of externally observable events, one cannot check whether the `transmission_failed` signal is being sent in response to a failed transmission, or is being sent as a constant response. Where necessary, the task of checking that the external behaviour of the system conforms to internal events is handled in this design methodology using temporal logic.

3.7.2 Validation Using Temporal Logic

Temporal logic formulae are checked against the system with the LMC model checker (see section 3.5.3). Although it might be desirable to check formulae against the unrestricted state space expansion of the system specification, this practice has proved impractical for all but the simplest of specifications. Instead, the specification is composed in parallel with a test sequence to limit the state space expansion, and the LMC model checker is then used to test the formula against only that restricted subset of the state space. If the formula is to be used to check the internal behaviour of the system, the system specification may need to be rewritten slightly to remove gate hiding, such as in the example below:

```
process communication_link[s, t, l]:noexit:=
    sender[s, l]
    |[l]|
    receiver[t, l]
endproc
```

Even if test sequences are used to reduce the state space expansion, it is possible that the expansion cannot be completed within a reasonable time. The LMC model checker allows the user to specify the maximum width and depth of expansion, and then reports whether the formula specified can be said to always hold, or to hold only to the depth explored.

3.8 Implementation of the Specification

Given a validated LOTOS specification, written in the style presented in the preceding sections, it is now possible to derive a ROOM model of the specified system. This section presents a mapping from the implementation-oriented LOTOS style to the ROOM notation. The fact that the LOTOS specification has been validated against requirements, together with the clear mapping from LOTOS to ROOM, should yield confidence in the correct operation of the ROOM model and hence the C++ implementation. Further confidence in the ROOM model may be gained through the use of the same agent scenarios as used in the validation of the LOTOS specification (see section 3.9.6).

3.8.1 Actor Structure Implementation

The ROOM notation is based upon the use of actors that encapsulate their internal data and behaviour and communicate only through defined sets of ports. The LOTOS fragment below describes an actor, `P`, that interacts with other actors only through the gate `hR1`. Hidden within actor `P` are two contained actors, `P1` and `P2`. These two actors communicate through a hidden gate, `gR1`, and actor `P1` communicates with the outside world through gate `hR1`. Given this LOTOS actor specification, the ROOM representation illustrated in figure 3.3 can be derived. Note how the gates in the LOTOS specification map to the ports in the ROOM notation; actor `P1` has two gates, `gR1` and `hR1`, which map to the two ports seen in the ROOM model. Also note how the hidden communication channel `gR1`, represented in LOTOS through the `hide` operator and the generalised parallel operator (`|[gR1]|`), is represented in ROOM as the link between the two actors `P1` and `P2`, which is not visible outside the actor `P`. Finally, note that the ROOM relay port `hR1` is derived from a gate on

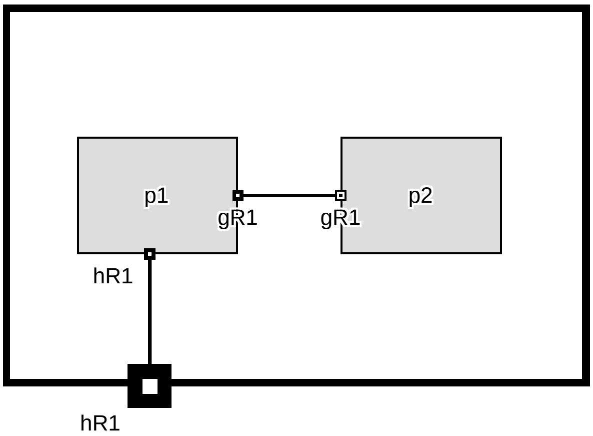


Figure 3.3: ROOM notation for a hierarchically-structured actor

a contained actor (the gate `hR1` on actor `P1`) that is not hidden and that appears in the gate list of the containing actor `P`.

```
process P[hR1] : noexit :=
    hide gR1 in
    (
        P1[gR1, hR1]
        |[gR1]|
        P2[gR1]
    )
where
    process P1[gR1, hR1] : noexit :=
        . . .
    endproc
    process P2[gR1] : noexit :=
        . . .
    endproc
endproc
```

For each of the gates of an actor, a protocol class should be defined in ROOM. The protocol class specifies the possible types of messages that may be passed through a gate. Each of these types is defined as a data class in ROOM. Because, at present, this methodology deals only with enumerated data types, all data classes will be of type *Enumerated*. The values of these enumerated data classes may be determined from the enumerated type definitions in the LOTOS specification. For example, the LOTOS type definition below indicates that the message can take one of two values, `ACTIVATE_request` or `DEACTIVATE_request`. Thus, an enumerated data class is defined in ROOM with the two values as indicated. Then, assuming that these are the only possible outgoing message values on gate `Ss`, a protocol class is defined such that the outgoing messages are of the data class just defined. The port `Ss` is then an instance of this protocol class.

```
type Ss_out is
    sorts Ss_out
    ACTIVATE_request, DEACTIVATE_request :-> Ss_out
endtype
```

A more complex example of this comes from the alternating-bit protocol example seen later in this chapter (section 3.9 onwards). A data class composed of two components, a number and a bit, is defined using the LOTOS data type definition below:

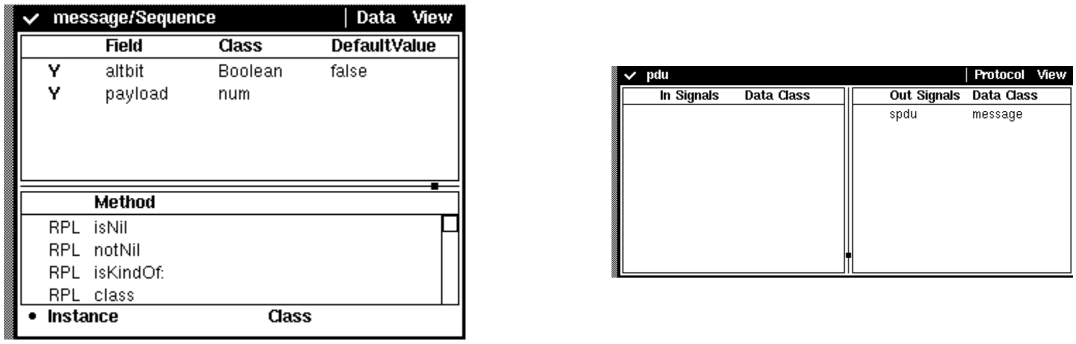
```
type message is number, altbit
sorts mess
opns data :  mess -> num
     seq :  mess -> bit
     msg :  num , bit -> mess
eqns forall Data:  num, Seq:  bit
   ofsort num
     data(msg(Data,Seq))= Data;
   ofsort bit
     seq(msg(Data,Seq)) = Seq;
endtype
```

In the ROOM implementation, a corresponding data class is defined as in figure 3.4(a). A protocol class is then defined, using the data class `message`, as in figure 3.4(b).

3.8.2 Actor Behaviour Implementation

The LOTOS style for specification of behaviour described in section 3.6.3 maps to the behavioural notation used within ROOM, ROOMcharts. ROOMcharts provide a powerful state machine notation, allow for hierarchical structuring of states, and permit the execution of action code on transitions, state entry or state exit.

A LOTOS actor that does not contain other actors will have its behaviour described as a state machine in the following format:



(a) ROOM data class definition (b) ROOM protocol class definition

Figure 3.4: ROOM data and protocol class definitions

```
process <process-name> <gate-list> <value-list> :  noexit:=
    <initial-behaviour> (optional)
    <initial-state><gate-list>
where
    process <state-1> <gate-list> <value-list>:  noexit:=
        ...
    endproc
    process <state-2> <gate-list> <value-list>:  noexit:=
        ...
    endproc
endproc
```

In the format above, the optional `<initial-behaviour>` describes actions that occur before entering the initial state of the actor; these actions will typically be used for initialisation. The `<initial-state>` must be one of the process names listed as `<state-1>`, `<state-2>` and so on. This description of the actor maps to the ROOMcharts representation of actors, in which an initial transition is made without a triggering event (see the top left corner of figure 3.6(a) for an example of this initial transition). The actor then remains in the initial state until a triggering event causes a transition to another state. The `<value-list>` is optional, and is used to specify extended state variables. As such, the list must be the same for each process.

Transitions to other states may be described in three main ways in LOTOS. The simplest case is that of a state that only has one possible outgoing transition, and is specified as in the example below:

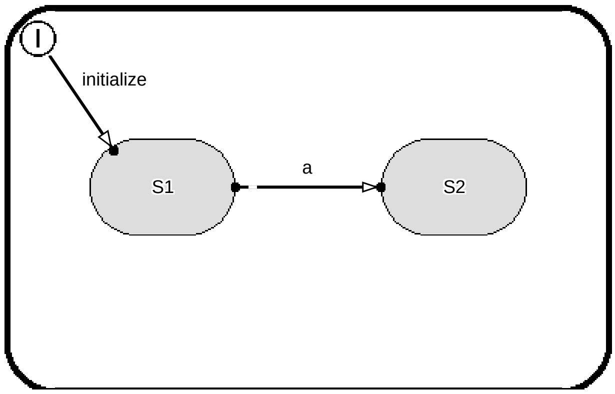


Figure 3.5: ROOM behaviour diagram for a simple transition from one state to another.

```
process S1[gate-list-S1]:noexit:=
    a ! msg_value_1;
    S2[gate-list-S2]
endproc
```

This LOTOS fragment illustrates a state `S1` with an outgoing transition that is triggered by a message on gate `a` with value `msg_value_1`. If such a message arrives, the transition is triggered and the actor moves to state `S2`. In ROOM, this case would be represented by a single transition from actor `S1` to `S2`, with the port `a` listed in the list of triggering events for the transition, and a test for the value `msg_value_1` as a guard condition on the transition (see figure 3.5).

The second case considered is that of a state with an outgoing transition that branches according to the value of the triggering message. The LOTOS fragment below illustrates a state `S1-2` with an outgoing transition triggered by the arrival of a message on gate `a`. If a message arrives on that gate, the transition is always taken, though the destination of the transition depends upon the value of the message. If the message value is `true`, the next state is `S1-4`, while if the message value is `false`, the next state is `S1-5`.

```
process S1-2[gate-list-S1]:noexit:=
    a ?  msg:message_type;
    (
        [msg = true] -> S1-4[gate-list-S1]
    []
        [msg = false] -> S1-5[gate-list-S1]
    )
endproc
```

This LOTOS state description maps to the ROOM notation seen in figure 3.6(a), in which the choice point tests the value of the message. The graphical representation of the transition must be augmented by a textual description of the triggering event, entered into the list of triggers for the transition in the ObjecTime toolset. The choice point has a fragment of C++ code associated with it to test the value of the message. The C++ code must return a boolean value indicating which branch is to be taken. An example of such a code fragment might be:

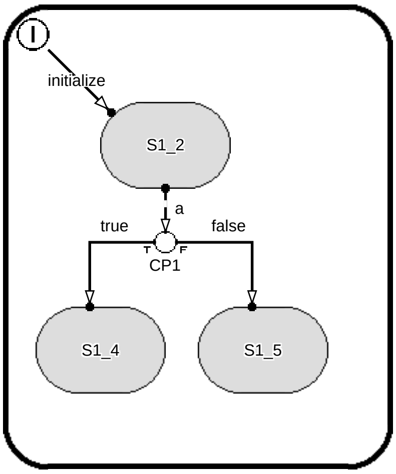
```
return (*RTDATA == "true");
```

The ObjecTime system provides the macro `RTDATA` that points to the value of the last message received. Thus, this C++ statement returns the boolean `true` if the last message received was `"true"` and `false` otherwise.

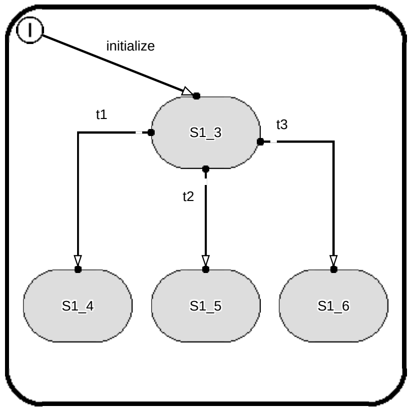
The last case considered is that of a state with multiple outgoing transitions that may be triggered by different messages. The transitions are separated by the choice operator (`[]`) in LOTOS, as in the LOTOS fragment below.

```
process S1-3[gate-list-S1]:noexit:=
    a !  msg_value_1;
    S1-4[gate-list-S1]
    []
    a !  msg_value_2;
    S1-5[gate-list-S1]
    []
    a !  msg_value_3;
    S1-6[gate-list-S1]
    ...
```

This state specification maps to the ROOM notation seen in figure 3.6(b). Each of the transitions in the ROOM model must be annotated with the triggering event (here, a message arriving through port `a`) together with a guard condition. The guard condition indicates that the transition can only be taken if the condition is satisfied. In this case, the guard condition for each transition would test the contents of the message.



(a) State diagram illustrating two-way choice



(b) State diagram illustrating multi-way choice

Figure 3.6: State diagrams derived from LOTOS behaviour specifications.

The discussion of behaviour above has dealt only with simple transitions between states, triggered by incoming messages. In order that actors be able to perform behaviour, actions must be associated with transitions or states. Generally speaking, these actions will consist of sending messages to other actors, and the issue of communication is discussed in the following section.

3.8.3 Communication Implementation

As indicated earlier, LOTOS specifications may require synchronous or asynchronous communication between actors. Synchronous communication is simply indicated using the LOTOS rendezvous mechanism, while asynchronous communication is specified using an intermediary process to represent the communication channel. Communication is specified in the ROOM notation by attaching C++ code actions to transitions or states. Thus, one can specify that if a given transition is taken, a message should be sent through a given port.

If synchronous communication is required, the ROOM *invoke* primitive is used. A simple synchronous communication might be specified in LOTOS as:

```
aPort ! signal ! dataObject;
```

The gate through which communication occurs is `aPort` and the communication carries two values, `signal` and `dataObject`. This synchronous communication can be represented in the ROOM notation as:

```
retMessagePtr = aPort.invoke(signal, dataObject);
```

The port through which the communication should occur (corresponding to the gate in the LOTOS specification) is specified by `aPort`, while the signal type is indicated by `signal` and the payload by `dataObject`. The recipient actor would be specified in LOTOS with an action:

```
aPort ! signal ? data:data_type;
```

In ROOM, the recipient actor must include a *reply* operation, which may be of the form:

```
msg->reply(signal);
```

The `reply` method synchronises with the `invoke`, and data exchange occurs. Optionally, data may be passed in both directions. This would be specified in LOTOS as:

```
aPort ! signal ! dataObject ? reply:reply_type;
|[aPort]|
aPort ! signal ? data:data_type ! replyObject;
```

Correspondingly, the ROOM notation supports an extended form of the reply operation to specify that data is to be returned:

```
msg->reply(signal, replyObject);
```

Asynchronous communication is provided via the *send* primitive, typically of the form:

```
aPort.send(signal, priority, dataObject);
```

As above, `aPort` indicates the port from which the message will be sent, `signal` is a symbolic name for the message, `priority` allows for priority ordering of triggers and `dataObject` is a container for the data payload of the message. For the purposes of the work described here, it will be assumed that all messages are of normal priority, so the only features of concern are the gate, the signal and the payload.

If the recipient actor is in a state to receive the message, the message will trigger a transition and the contents of `dataObject` can be assigned to an internal variable or tested as part of a guard condition or choice point.

It should be noted that the semantics of LOTOS and ROOM communication differ in their degree of coupling. If asynchronous communication is used between ROOM actors, they may be regarded as very loosely coupled. An actor wishing to send a message out on a port may do so without there being an actor ready to receive the message. That is, if actor A sends a message through a port that is connected to actor B, it is not necessary that actor B do anything with the message. If the current state of the recipient does not have a transition triggered by a received message, the message is simply discarded. An advantage of this communications semantics over a more tightly coupled system is that the sender will never block if the recipient cannot handle the message. However, this loose coupling may mask design errors. If the designer intends that a message should affect the recipient, but the recipient actor does not have a transition triggered by that message, the ROOM model will still execute without indicating an error. Thus, detection of "unspecified reception" is not possible in the ROOM semantics (see [ZWR+80] for a discussion of unspecified reception).

The LOTOS specification style outlined in this chapter provides differing levels of coupling. If two actors are connected by a communication channel that cannot lose messages, such as the example on page 55, then if the recipient cannot handle an incoming message, the system will deadlock. While this would not be desirable in an implementation, the presence of deadlocks alerts the designer to possible problems in the design. If the designer decides that it should be possible for the message to arrive when there is no sensible action to be taken, the LOTOS specification can be augmented to include a synchronisation with the message in order to dispose of it.

If, however, two LOTOS actors are connected by a communication channel that can lose at most one message, such as the example on page 56, then if the recipient cannot handle an incoming message, the channel will simply synchronise with the next incoming message and discard the first one. This channel specification would only allow for one ignored message; in order to allow for an arbitrary number of unlimited messages, the channel would also need to be able to lose an arbitrary number of messages, as seen in the example on page 56.

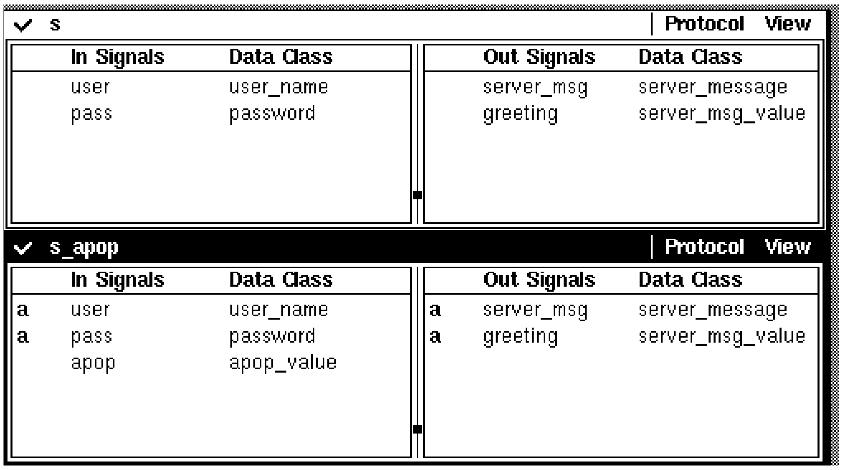
Providing these varying levels of coupling yields flexibility during the design stage; if the designer wishes to be alerted to possible unspecified receptions, a tightly coupled channel is chosen. If, on the other hand, the designer wishes for the specification to execute without blocking on ignored messages, a channel that loses messages may be selected. At present, this variable level of coupling is connected to possible loss of messages. Further research could explore the possibility of allowing for looser coupling between actors without also modelling the loss of messages in the communication channel (see on page 129).

3.8.4 Inheritance Implementation

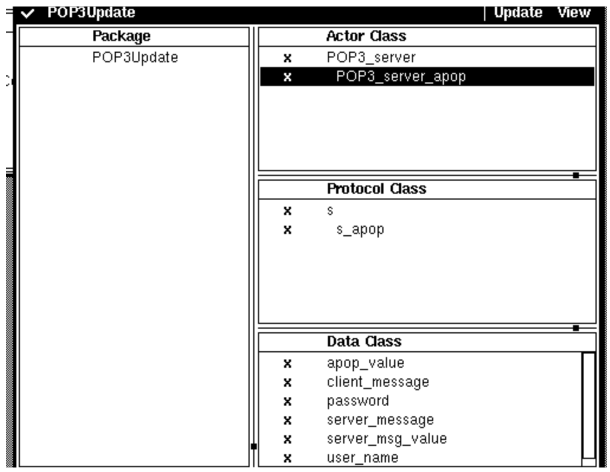
ROOM provides inheritance mechanisms for data, protocol and actor classes. These mechanisms may be used to implement an inheritance hierarchy originally specified in LOTOS. Section 3.6.4 described a LOTOS notation that could be used to specify that one LOTOS actor inherited behaviour from another. For example, the process specification below indicates that process `D` inherits from `B`, while adding new behaviour and states.

```
process D[gates-D](parameters-D):exit-behaviour-D:=
extends process B[gates-B](parameters-B):exit-behaviour-B:=
    initial-behaviour-D
where
    process D1[gates-D1](parameters-D1):exit-behaviour-D1:=
    extends process B1[gates-B1](parameters-B1):exit-behaviour-B1:=
        behaviour-D1
    endproc
endproc
```

Section 3.6.4 also indicated the conditions that must apply to the definition of process `D` so that it can inherit from `D`. For example, the gate parameter list `gates-B` must be a prefix (proper or not) of `gates-D`. If gates have been added, corresponding protocol classes must be defined in ROOM and added to the definition of the actor class `D`. If a given gate is expected to pass additional messages in actor `D`, the corresponding protocol class must be extended (see example in figure 3.7(a)). This extension is performed by creating a new protocol class that inherits from the old one, and adding the new messages to the new class. Similarly, if a given message type should include extra messages, a new data class is created that extends the class enumerating the old messages. A new protocol class must then be defined to use this new data class. Finally, a new actor class is defined that inherits from the old class, and uses the new protocol classes. Figure 3.7(b) illustrates how inheritance is indicated in ROOM using indentation. For example, protocol class `s_apop` is indented with regard to its base class `s`. New behaviour is defined by adding states to the behaviour definition of the actor. Figure 3.8 illustrates how inheritance in behaviour definitions is indicated in ROOM. Note that states that are inherited from the base class are bordered in gray in the derived class (for example, the state `waiting_for_password`).

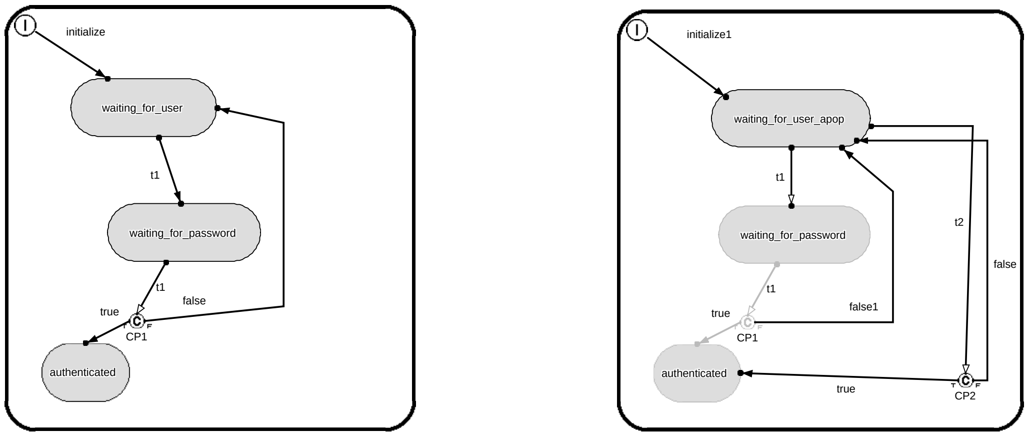


(a) Example of inheritance in ROOM protocol classes.



(b) Representation of inheritance in ROOM.

Figure 3.7: Examples of inheritance in ROOM.



(a) Behaviour description for the base actor class.

(b) Behaviour description for the derived actor class.

Figure 3.8: Example of inheritance of behaviour in ROOM.

3.9 An Illustrative Example: The Alternating Bit Protocol

In order to illustrate this discussion, the process was applied to the specification and implementation of a simple example protocol. The example chosen is sometimes referred to as Stop-and-Wait ARQ (see Bertsekas and Gallager [BG92], page 66). This protocol is used to ensure reliable communication of data packets by guaranteeing that each packet has been received correctly before attempting to transmit the next. Thus, the protocol may be described at a system level by requiring that the sequence of messages observed by the receiver be the same as the sequence injected into the sender (see figure 3.9). The remainder of this section will explain the operation of the protocol, concluding with the system requirements used in the creation of the specification. The following sections (3.9.1 onwards) will explain the work undertaken to produce a LOTOS specification in the style proposed in this chapter, to formally validate the specification and to derive a ROOM implementation of the protocol. A more detailed explanation of the specification and implementation is provided in appendix A.

To ensure that a packet has been received, the sender waits for an acknowledgement from the receiver. However, if this acknowledgement is undistinguished, it is impossible for the sender to know whether the receiver was acknowledging the last

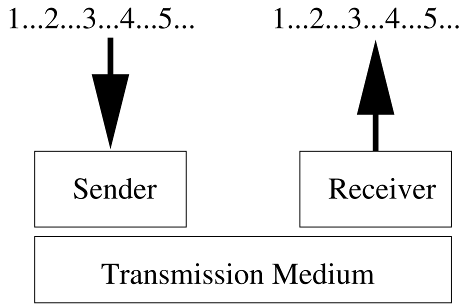
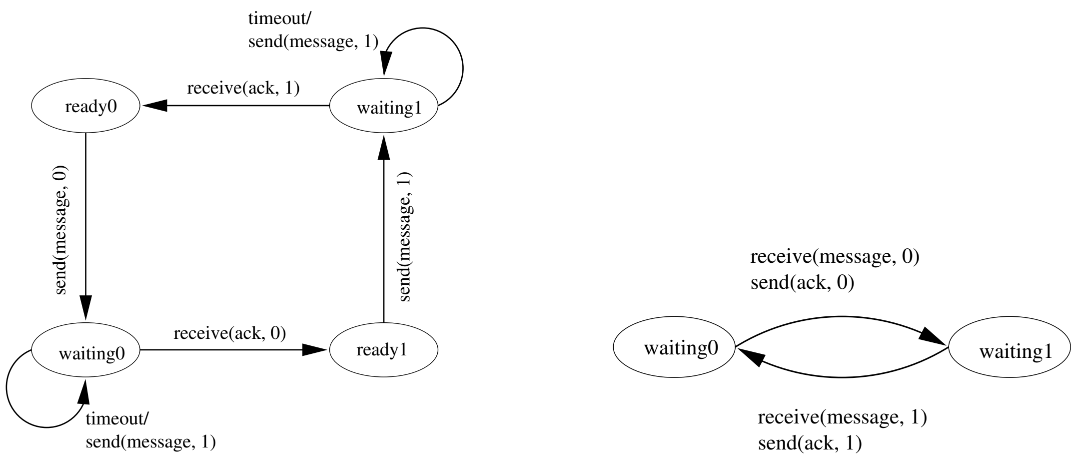


Figure 3.9: System-level description of the alternating bit protocol

packet sent, or whether the acknowledgement was simply a late acknowledgement of a previous packet. To counteract this potential problem, Stop-and-Wait ARQ uses sequence numbers attached to a transmission and requires that the acknowledgement similarly include a sequence number. By checking these numbers against the number expected, both the sender and the receiver can be sure that the packet or acknowledgement that they have just received is valid. It has been shown by Bartlett et al. that reliable transmission over a queue-like medium can be achieved if the sequence number is modulo 2 [BSW69]. That is, we need only use a single bit sequence number and alternate between the values 0 and 1. For this reason, the protocol is sometimes called *alternating bit protocol* (ABP).

The operation of the system is described by the state diagrams in figure 3.10. The sender (illustrated in figure 3.10(a)) starts in state `ready0`. When the sender has a message to be sent, it sends the message to the receiver, along with the alternation bit 0 and moves to the state `waiting0`, waiting for an acknowledgement with the alternation bit set to 0. Acknowledgements with the wrong alternation bit cause the message to be resent, and the message may also be resent if an acknowledgement is not received within a reasonable time period. Once a correct acknowledgement is received, the sender moves to state `ready1`. If it now has a message to be sent, it packages the message with the alternation bit 1 and moves to the state `waiting1`, waiting for an acknowledgement with the alternation bit set to 1. Upon reception of a correct acknowledgement, the sender returns to state `ready0` and the cycle restarts.

The receiver (illustrated in figure 3.10(b)) is a simpler state machine, as it needs only to maintain a record of the alternation bit which it is expecting. The two states `waiting0` and `waiting1` indicate the expected alternation bit. When a message arrives, an acknowledgement is sent with whatever alternation bit was attached to the message. If the alternation bit is as expected, the message is a new one and is passed to the upper layers of the system. If the alternation bit is incorrect, the message must be a retransmission of a previously received message, and the message is discarded.



(a) State diagram for sender

(b) State diagram for receiver

Figure 3.10: State diagrams for the alternating bit protocol

Because ABP is simple, the system requirements are also simple. The protocol is intended to allow reliable transmission of data packets, and so requires that the sequence of messages presented to the sender be received by the receiver in the order presented, as in figure 3.9.

3.9.1 Formalisation of Requirements

The thesis used the above system description of the alternating-bit protocol to test the design methodology. The simplicity of the system requirements meant that the creation of agent scenarios typically involved ensuring that a sequence of messages be received at the other end in the same order. However, this simple agent scenario was somewhat complicated by the existence of an unreliable transmission medium. While the use of a simulated reliable medium was an important check to ensure that the protocol in principle worked, the purpose of the alternating-bit protocol is to overcome the problems of an unreliable medium, and so the requirements had to specify system behaviour in the context of an unreliable medium. It is important that a medium that occasionally loses either messages or acknowledgements does not prevent a sequence of messages from being received in order. However, if the medium consistently loses messages, as it might do if a physical line were cut, there will be no sequence of received messages, and so the naive scenarios would not have adequately captured this failure.

The informal requirements above were separated into safety and liveness requirements (see section 3.5.1). The main safety requirement was that for a sequence of input messages numbered 1, 2, 3 and so on:

1. When message n arrived, all messages numbered 1, 2, ..., $n-1$ had already arrived.
2. When message n arrived, its predecessor was message $n-1$.

This second requirement may be seen as recasting the first requirement in an inductive form, so that if the second was true, the first would also be true. Formulating the second requirement in CTL, using the Precedence Property Pattern described by Dwyer et al. (see section 3.5.3), yielded the formula:

$$A(\neg P\,\mathcal{U}\,(S \vee AG(\neg P)))$$

The proposition P states that message n has arrived and proposition S states that message $n-1$ has arrived.

Similarly, temporal logic was used to specify the liveness property that progress will be made. Again, following Dwyer et al., the requirement that if message n was received, then message $n+1$ would be received at some point in the future was written as:

$$AG(P \longrightarrow AF(S))$$

Proposition P indicates that message n was received, and proposition S indicates that message $n+1$ was received.

The formalisation of requirements was completed by combining agent scenarios with temporal logic assertions. Thus, the scenarios specified the normal behaviour of the system in terms of sequences of sent and received messages, while the temporal logic assertions expressed requirements for safety and liveness.

3.9.2 Development of Specification

Given the system description and formalisation indicated above, a specification was developed, using the methodology outlined earlier in this chapter. The specification was validated against the requirements formalised in the preceding section, and then used to derive a ROOM implementation. The subset of LOTOS used in the alternating bit specification differs slightly from that presented in section 3.3 in using structures composed of enumerated types. Using structures allowed for a shorter and more comprehensible specification than using two data items on each communication, but does not greatly affect the illustration of the methodology.

The alternating bit protocol is a relatively simple protocol, though it allowed for a certain amount of hierarchical decomposition. The first level of decomposition rested on the physical separation of the sender and the receiver, and so yielded the skeleton specification below. A process representing the channel was included to specify message loss in the specification; in the implementation there was not any code representing the channel as such.

```
process abp_system[sendmsg,recvmsg]:noexit:=
    hide sendpdu,recvpdu,sendack,recvack in
    (
        abp_s [sendmsg,sendpdu,recvack]
            |[sendpdu,recvack]|
        channel [sendpdu,recvpdu,sendack,recvack]
            |[sendack,recvpdu]|
        abp_r [recvmsg,sendack,recvpdu,r_test]
    )
endproc
```

The hide operator was used in this fragment to hide the internal workings of the ABP. Because gates such as `sendpdu` were hidden, the only externally visible gates were `sendmsg`, used to specify the data to be sent, and `recvmsg`, used to make the data received from the sender available. This specification was decomposed further because ABP requires a timer to determine when an unacknowledged packet should be retransmitted. The sender was decomposed as below:

```
process abp_s[smsg, spdu, rack] :  noexit :=
    hide tsig in
    (
        abp_s_state_machine[smsg, spdu, rack, tsig]
            |[tsig]|
        abp_s_timer[tsig]
    )
endproc
```

Again, the LOTOS hide operator was used to encapsulate the internal workings of the sender. The existence of the timer was invisible from outside the hide operator. The external structure and behaviour of the sender remained the same, and further decomposition could therefore have been carried out without affecting the externally visible characteristics of the sender, as long as any internal gates were hidden.

The channel was specified as a single process that consisted of two processes interleaving, thus representing the two directions of transmission.

```
process channel [in1,out1,in2,out2] :  noexit :=
    chann [in1,out1]
        |||
    chann [in2,out2]
endproc
```

As described in section 3.6.5, an unreliable channel could be specified in such a way that it could lose an arbitrary or a bounded number of messages. For the purposes of the alternating-bit protocol, a bounded specification was favoured because it prevented excessive state space expansion. Bounded message loss allowed verification that the protocol could arrange for retransmission without waiting too long for the verification tool to complete. Although the channel specification described in section 3.6.5 allowed only one message to be lost, it would have been trivial to expand this specification to allow for a larger bound on the number of lost messages before successful transmission. Both specifications of the channel were used during validation of the specification (see section 3.9.3).

The specification of the sender and receiver was achieved through the LOTOS style presented earlier, which represented each state as a process. Because the behaviours of the sender and receiver were so simple, there was no hierarchy of states; the sender could be in one of four states, while the receiver could be in one of two states. The sender had already been decomposed into a state machine and a timer, as illustrated above. The state machine was specified as:

```
process abp_s_state_machine[smsg,spdu,rack,tsig]:noexit:=
    sready0[smsg,spdu,rack,tsig](zero of num) (* initial state *)
where
    process sready0[smsg,spdu,rack,tsig](n:num):noexit:=
        smsg ?n:num;    (* receive message from upper layer *)
        spdu !msg(n, 0 of bit);         (* send to receiver *)
        tsig !timerStart;               (* start the timer *)
        (* move to a state in which we wait for acknowledgement *)
        swaiting0[smsg,spdu,rack,tsig](n)
    endproc
    process sready1[smsg,spdu,rack,tsig](n:num):noexit:=
        smsg ?n:num;
        spdu !msg(n, 1 of bit);
        tsig !timerStart;
        swaiting1[smsg,spdu,rack,tsig](n)
    endproc
    process swaiting0[smsg,spdu,rack,tsig](n:num):noexit:=
        (
```

```
                rack ?X:mess;       (* receive acknowledgement *)
                (
                    [seq(X) eq 0 of bit] -> (* what we expected?  *)
                        tsig !timerStop;    (* stop the timer *)
                        sready1[smsg,spdu,rack,tsig](zero of num)
                []
                    [seq(X) ne 0 of bit] -> (* not what we expected *)
                        spdu !msg(n, 0 of bit);
                        tsig !timerStart;
                            (* continue to wait *)
                        swaiting0[smsg,spdu,rack,tsig](n)
                )
            []
                tsig !timeout;              (* timer period over *)
                spdu !msg(n, 0 of bit);
                tsig !timerStart;           (* restart the timer *)
                    (* continue to wait for acknowledgement *)
                swaiting0[smsg,spdu,rack,tsig](n)
            )
        endproc
        process swaiting1[smsg,spdu,rack,tsig](n:num):noexit:=
            ...similar to above...
        endproc
    endproc
```

Following the description of the protocol in the preceding sections, the sender was specified so that, when in either of the ready states (`sready0` and `sready1` above), it could accept data from the upper layers of the system (synchronisation on gate `smsg`). This data would then be encapsulated in a message with the current value of the alternation bit and sent to the receiver (the `spdu`). The sender would start a timer and move to a state indicating that it was waiting for an acknowledgement.

In the waiting states (`swaiting0` and `swaiting1`), the sender could receive an acknowledgement from the receiver. If the alternation bit in the acknowledgement was the one expected, the sender would stop the timer and move to a ready state. If the alternation bit was incorrect, the receiver was acknowledging a previous message. The sender would resend the current message and return to waiting for an appropriate acknowledgement. Alternatively, the sender could receive a timeout signal, indicating that an acknowledgement had not been received in the appropriate interval. The sender would resend the message and restart the timer.

The receiver consisted of only two states and did not interact with a timer. In both states the receiver was expecting a message from the sender; the two states differed only in the value of the alternation bit expected. If a message was received, the protocol required that it was always acknowledged (the `sack` in the specification below). If the alternation bit in the message was the same as expected, the transmission represented a new message and its data would be passed to the upper layers of the system (the `rmsg`) and the receiver moved to the other state, expecting a message with the other possible value of the alternation bit. If the alternation bit was not the same as expected, the transmission was a repeat of a previously received message, indicating that the sender had not received the acknowledgement in time. While an acknowledgement was sent, the receiver did not pass the data to the upper layers and remained in the same state. This behaviour is illustrated by the specification of the `rwaiting0` state below:

```
process abp_r[rmsg,sack,rpdu]:noexit:=
    rwaiting0[rmsg,sack,rpdu]          (* initial state *)
where
    process rwaiting0[rmsg,sack,rpdu]:noexit:=
        rpdu ?X:mess;                  (* receive message *)
        sack !msg(zero, seq(X));       (* always acknowledge *)
        (
            [seq(X) eq 0 of bit] ->            (* as expected?  *)
                rmsg !data(X);  (* yes, send data to upper level *)
                rwaiting1[rmsg,sack,rpdu]
        []
            [seq(X) ne 0 of bit] ->            (* not as expected *)
                rwaiting0[rmsg,sack,rpdu]      (* keep waiting *)
        )
    endproc
    process rwaiting1[rmsg,sack,rpdu]:noexit:=
        ...similar to above...
    endproc
endproc
```

Finally, the timer was specified as being in one of two states, either ready to start timing (`timerready`) or actually timing an interval (`timertiming`). If the timer was in the ready state and received a `timerStart` message, it made the transition to the `timertiming` state, modelling the commencement of a timer interval, while if it received an irrelevant `timerStop` message, it ignored it and remained in the `timerready` state, as seen in the specification of the `timerready` and `timertiming` states below:

```
process timerready[tsig] : noexit :=
    (
        tsig !timerStart;
        timertiming[tsig]
    []
        tsig !timerStop;
        timerready[tsig]
    )
endproc

process timertiming[tsig] : noexit :=
    (
        tsig !timerStart;
        timertiming[tsig]
    []
        tsig !timerStop;
        timerready[tsig]
    []
        tsig !timeout;
        timerready[tsig]
    )
endproc
```

The final specification contained a little over 200 lines of LOTOS, not including test sequences. Once the specification was completed, it was then possible to use the tools of the XELUDO toolkit to compile the specification to an executable form and to use the interactive simulator, ISLA, to perform simple checking (thus ensuring that the specification behaved as expected and that there were no obvious deadlocks), before moving on to the formal validation.

3.9.3 Validation

As described in section 3.7, the LOTOS specification was validated using validation sequences derived from agent scenarios, and by model checking of temporal logic formulae.

Validation Using Agent Scenarios

A simple black box validation sequence was written as illustrated below. The sequence represented the system being used to send a short sequence of numbers, consisting only of the values `one` and `two` (enumerated values were used rather than the `Natural` data type to simplify the simulation process). Correct system operation required that the same sequence, in the same order, be observed at the other end. Note that the process `abp_tester`, which was composed in parallel with the alternating bit system, synchronised only on the gates `sendmsg` and `recvmsg`, so it did not examine the internal behaviour of the system. Also, the use of two interleaved processes (`abp_tester_s` and `abp_tester_r`) meant that the ordering of events between the two ends of the system was not important. That is, even if both messages were sent before the first one was received, the system still conformed to requirements. This test sequence provided a more flexible validation than if a global ordering on events had been set (for example, requiring that the first message had been correctly received before the second one was sent). The test sequence is illustrated below, indicating how the two processes were interleaved:

```
abp_system[sendmsg,recvmsg]
|[sendmsg,recvmsg]|
abp_tester[sendmsg,recvmsg]

process abp_tester[sendmsg, recvmsg] : noexit :=
    (
    abp_tester_s [sendmsg]
    |||
    abp_tester_r [recvmsg]
    )
    >>
    success;
    stop
endproc

process abp_tester_s [s] : exit :=
    s ! one of num;
    s ! two of num;
    exit
endproc

process abp_tester_r [r] : exit :=
    r ! one of num;
    r ! two of num;
    exit
endproc
```

The `abp_tester` process included the special event `success`, used by the LOLA tool to indicate successful execution: if an execution trace reached this action, it would stop and report success. When the system above was executed using LOLA, the tool reported a *may pass*[7] result, having analysed 266 states, generated 438 transitions and found 115 successes. One execution path was truncated, indicating that it had not reached a `success` event within the specified expansion depth. This excessively long execution path represented the repeated loss of messages by the unreliable channel.

Validation Using Temporal Logic

As with the validation sequences, the temporal logic validation of the system was carried out at two different levels. Testing the system against temporal logic formulae created during the requirements analysis (see section 3.9.1) provided a form of black box temporal logic validation, while using formulae that specified temporal ordering of internal events provided grey box temporal logic verification.

The temporal logic validation and verification were performed using the LMC model checker. This tool takes a compiled specification and allows the designer to enter formulae to be checked. The model checker was used on two versions of the specification:

1. One version used the first definition of the channel in which the channel could continually lose messages.

2. A second version used the second definition of the channel in which the channel could lose at most one message.

For the first version of the specification, the safety requirement that messages arrive in order was specified by mandating that the predecessor of message n was message $n-1$, expressed as:

$$A(\neg P\,\mathcal{U}\,(S \vee AG(\neg P)))$$

The proposition P states that message n has arrived and proposition S states that message $n-1$ has arrived. The LMC model checker reported that this formula 'holds to the specified depth'[8], and so indicated that it could not expand the specification fully (keeping in mind that the continual message loss meant a potentially infinite

[7]See section 3.7.1 for an explanation of the results of LOLA validation.

[8]In the model checking of the ABP specification, an expansion depth of thirty transitions was selected to allow results to be obtained in a reasonable time.

state space), but that the formula held within the expansion that it was able to carry out.

This first version of the specification was also checked for the liveness requirement that a message injected at one end should arrive at the other. In CTL, this requirement is specified as:

$$AG((sendmsg!one) - - > EF(recvmsg!one))$$

This formula states that, whenever a message is sent, then it arrives at the other end in some future path. The formula cannot be strengthened to require that the message is delivered in all future paths, because the communication channel has been specified such that messages may continually be lost. The LMC model checker reported that this formula 'holds to the specified depth'.

The second version of the specification was validated against the following temporal logic formulae:

1. To check for liveness, data `two` must eventually be received. In CTL, this was represented as $EF(P)$ where P is the reception of the data (in the LOTOS of the specification, this was `ef(recvmsg!two)`). When the channel was specified in such a way that it did not continually lose messages, LMC was able to expand fully the state space, and so it confirmed that the formula held.

2. To check a safety requirement, the value `one` must arrive before the value `two`. LMC provides extensions to the regular CTL syntax that allow a more compact representation of this condition:

   ```
   ag(sendmsg!one --> all(recvmsg!one before recvmsg!two)).
   ```

   The use of the global quantification, `ag` requires that *whenever* there is a `sendmsg!one`, there must be a `recvmsg!one` before a `recvmsg!two`. It was not possible for LMC to determine whether this is always true, as doing so would require generation of an infinite state space. However, LMC could determine that the formula held within the state space explored, and reported the expression 'holds to the specified expansion depth'.

3.9.4 Implementation of the Specification in ROOM

The style of the LOTOS specification mapped naturally to a ROOM model. The two top-level processes, `abp_s` and `abp_r` were implemented as actors. The timer process was not realised in the ROOM model, though it could have been represented through ObjecTime's support for timing behaviour. The channel was implemented simply by linking the two actors. Within each actor, the LOTOS processes corresponding to each state were represented by states in the ROOMchart notation. The first action in each branch of LOTOS code was represented by a guard on a transition, and the remainder of the code was represented by C++ code entered into the transition editing window in the ObjecTime toolset. For example, the LOTOS specification of the receiver state `rwaiting0` included the following lines:

```
1    process rwaiting0[rmsg, sack, rpdu] :  noexit :=
2        rpdu?X:mess;
3        sack!msg(zero,seq(X));
4        (
5            [seq(X) eq 0 of bit] ->
6                rmsg!data(X);
7                rwaiting1[rmsg, sack, rpdu]
8                []
```

These lines of LOTOS described the transition from the `rwaiting0` state to the `rwaiting1` state, in the case in which a message was received and its alternation bit was as expected. The C++ representing this in the ROOM model is illustrated below.

```
INLINE_METHODS int abp_r_Actor::guard2_t1_event1()
{
return (((message*)msg->data)->bitsetting() == false);
}

INLINE_METHODS void abp_r_Actor::transition2_t1()
{
recvmsgR1.send(rmsg, ((message*)msg->data)->getpayload());
}

INLINE_METHODS void abp_r_Actor::exit2_rwaiting0()
{
rackR1.send(ack, message(0, ((message*)msg->data)->bitsetting()));
}
```

The three segments of code represented the following functions:

1. The first segment, indicated by the function definition `guard2_t1_event1()`, was the *guard* condition which determined whether the transition should be taken, and corresponds to line 5 of the LOTOS above. If this line of code returned *true* (in this case, if the alternation bit was correct), the transition was taken.

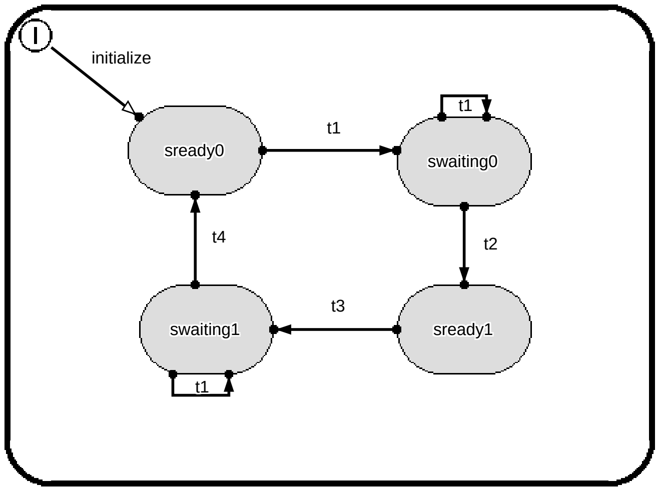


Figure 3.11: Behaviour specification for the ABP sender actor in the ROOM notation

2. The second segment, indicated by the function definition `transition2_t1()`, was the action code which should be executed when the transition was taken, and corresponds to line 6 of the LOTOS above. This code extracted payload data from the received message and passed it to the upper layer of the receiver.

3. The last segment, indicated by the function definition `exit2_rwaiting0()`, was the exit action code for the state, and corresponds to line 3 of the LOTOS above. This is code which was always executed when the state was left, regardless of the transition taken to enter the state. In this case, the receiver sent an acknowledgement to the sender containing the alternation bit extracted from the received message.

The implementation of the LOTOS specification continued in this vein, resulting in the ROOMcharts illustrated in figures 3.11 and 3.12. The two actors were linked through gates representing their connection via the channel, resulting in the system structure seen in figure 3.13. The ObjecTime toolset then allowed the model to be compiled, creating C++ code. The resulting C++ code was compiled to create an executable implementation.

3.9.5 Executing the ROOM Model

The ObjecTime toolset provides considerable facilities for executing compiled ROOM models. Figure 3.14 shows the toolset animating the ABP system. On the right is

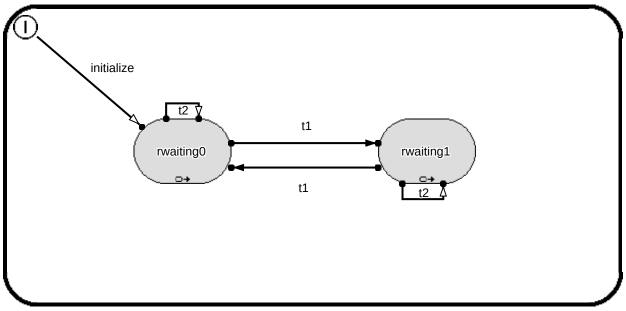


Figure 3.12: Behaviour specification for the ABP receiver actor in the ROOM notation

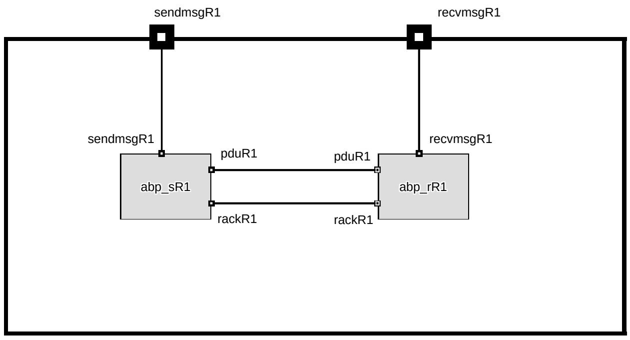


Figure 3.13: Structure of the ABP system in the ROOM notation

a state diagram of the sender process, `abp_s`. The box around the state `sready1` indicates that this is the current state of the sender, while the bold arrow leading into the process from the state `swaiting0` indicates that this is the most recent transition taken. Similarly, the state diagram for the receiver process indicates that the current state is `rwaiting1`. At the top of the screen are two windows that indicate the sequence of events at two ports, corresponding to the `sendmsg` and `recvmsg` gates in the specification. The contents of these two windows indicate that the messages sent (with data values 0, 1 and 2) were received in the same order. The left-hand column of each of the two windows indicates the time at which the message was detected, confirming for example, that the second message was sent (time 7221) after the first message was received (time 3774). The window just below the ObjecTime logo is used to specify messages which may be *injected* into the `sendmsg` port.

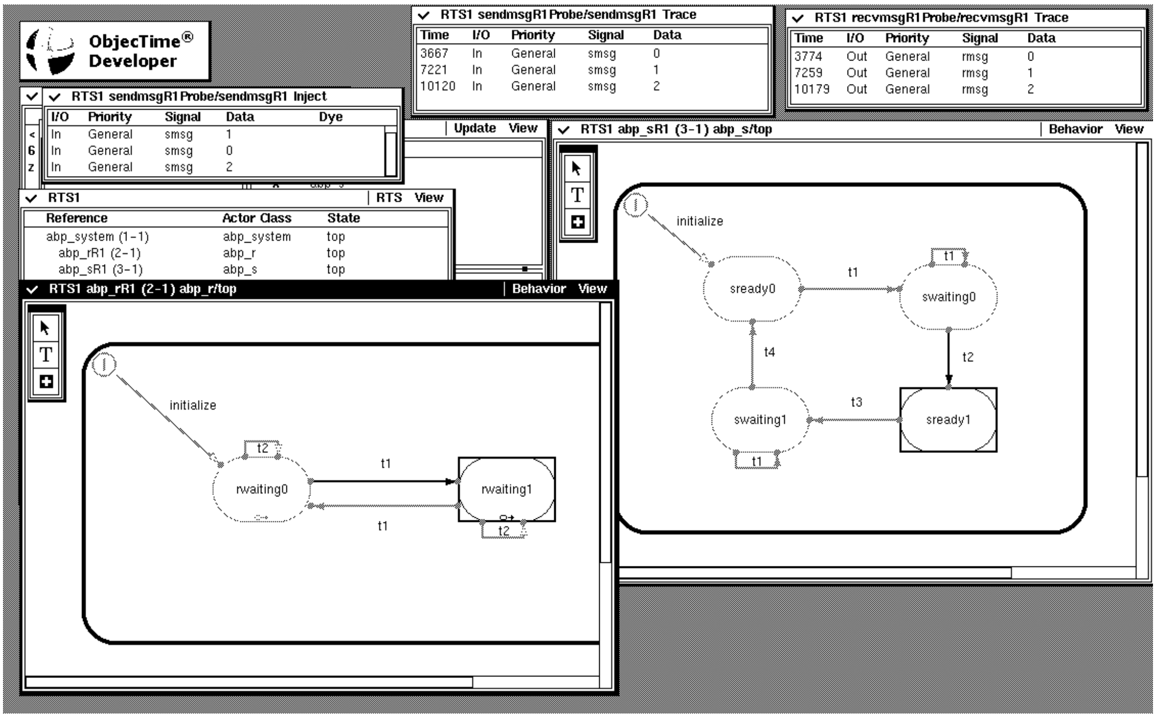


Figure 3.14: Screenshot of the ObjecTime toolset animating the ABP system

3.9.6 Validation of the ROOM Model

Although the derivation of the ROOM implementation from the validated LOTOS specification provides confidence in the correct operation of the implementation, it is still possible that errors may have been introduced in the manual translation. Further confidence may be gained by checking the ROOM implementation against the requirements. In this thesis, this checking has been carried out manually, using the debugging facilities of the ObjecTime toolset to, for example, inject messages into ports and observe the model's behaviour. By this means, some of the agent scenarios used in the validation of the LOTOS specification were reused in the validation of the ROOM implementation. Future research could examine the automation of this checking process, perhaps through enhancements to the ObjecTime toolset.

It should be noted that the validation of the ROOM model does not remove the need for validation during the development of the LOTOS specification. Validating the specification at various stages of its development allows for the early correction of errors, saving effort later in the system's development. If validation were left until the ROOM implementation had already been produced, it is possible that serious errors might lay undiscovered until this last stage, by which time correction would be difficult and expensive. Furthermore, detection of logical errors may be easier in LOTOS, a more abstract notation than ROOM, and LOTOS has available a wide range of validation tools such as LMC and LOLA. In particular, the ObjecTime toolset does not at present allow for state space expansion, preventing the use of model checking tools.

3.10 Summary

A design methodology was developed that includes a mapping from the LOTOS formal description technique to the ROOM modelling technique. In order to make this mapping process tractable, a subset of the ROOM notation was defined and this restriction placed limits on the LOTOS constructs that could be used. A subset of the LOTOS notation was defined, together with a specification style, so that the specifications produced should be amenable to mapping into ROOM and hence an implementation.

The use of LOTOS makes possible formal validation of design requirements, and two techniques were used in the methodology: validation sequences based on agent scenarios and temporal logic. Both agent scenarios and temporal logic formulae were created during the formalisation of system requirements, and then used for validation of the LOTOS specification.

The final stage of the methodology is the derivation of a ROOM model from the validated LOTOS specification using the mapping between LOTOS and the ROOM notation developed in this thesis. The ROOM model is validated manually using the same validation sequences as were used in the validation of the LOTOS specification. The ROOM model may then be compiled using the ObjecTime toolset to create a C++ implementation.

To illustrate the methodology, it was applied to a simple example: the alternating-bit protocol. The steps of the methodology were followed, resulting in an executable ROOM model.

Chapter 4

The GPRS Tunnelling Protocol

This chapter describes the application of the methodology presented in chapter 3 to the GPRS Tunnelling Protocol, to demonstrate the applicability of the methodology to an industrially-relevant protocol. Sections 4.1 and 4.2 provide background on the operation of the protocol, while the remainder of the chapter describes the application of the methodology to the protocol.

4.1 GSM and GPRS

GSM (the Global System for Mobile Communications) is a popular digital cellular telephony system, standardised by ETSI[1] in 1991, and commercially available since 1992 (see Mehrotra [Meh96] and Scourias [Sco95]). Since then, GSM has grown to attract more than 40 million subscribers in over one hundred countries. The system operates using time-division multiplexing (TDMA) in frequency bands around 900MHz (the original band assigned), 1.8GHz and 1.9GHz (referred to as DCS1800 and DCS1900, respectively). Since the first standard, development of functionality has continued, captured by phase 2 of the standard in 1996, and the upcoming phase 2+ standard. The phase 2+ standard includes enhancements to the data transmission services of GSM, such as the high-speed circuit-switched data service (HSCSD) and the general packet radio service[2] (GPRS).

The GSM network is composed of a number of cells, each served by a Base Transceiver Station (BTS). A number of BTSs are controlled by a Base Station Controller (BSC), which is responsible for ensuring that handoff occurs when a mobile user moves from one cell to another. A number of BSCs are connected to a Message Switching Centre (MSC), responsible for establishing connections and routing

[1] European Telecommunications Standards Institute

[2] See Brasche and Walke [BW97], Cai and Goodman [CG97] and the GPRS standard [gsm97]

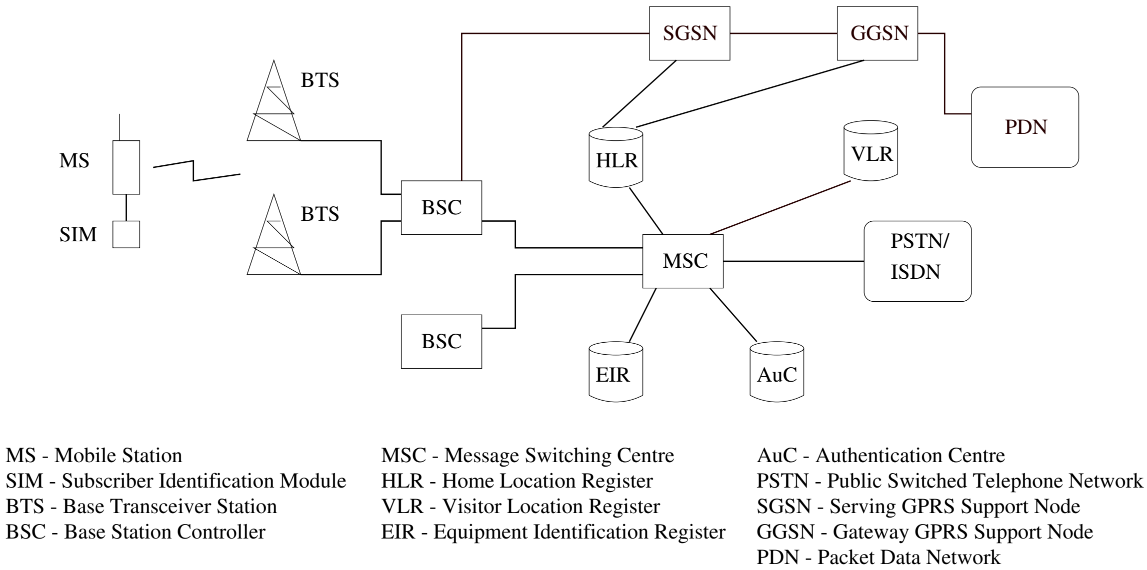


Figure 4.1: The GPRS network

calls. The MSC uses a number of databases to carry out its work, among them the Home Location Register (HLR), the Visitor Location Register (VLR), the Equipment Identity Register (EIR) and the Authentication Centre (AuC). The HLR stores subscription information for all network users, together with an indication of the last known location of a mobile user. Generally there will be only one HLR for a given network, and its data is used by all the MSCs in that network. Each MSC has its own VLR, which stores a copy of the subscription data for each user currently in the geographical area governed by that particular MSC. The EIR and AuC are used to authenticate respectively the mobile station equipment and the mobile user.

A call from a mobile user to a fixed user on the external telephone network is simply routed by the MSC to the external network. However, a call from one mobile user to another, or from a fixed user to a mobile user, must be routed to the correct area, and the location information in the HLR is used to indicate the last known location of the mobile user. The system uses this information to page the user within that area to establish the call. If the user's mobile phone is switched on, it will respond to the paging request and the call can be connected.

GPRS extends the GSM network by the addition of GPRS support nodes (GSNs) that route data transmissions to the GSM base station controllers (BSCs) (see figure 4.1). Serving GSNs (SGSNs) are responsible for routing transmissions to and from mobile stations within their serving areas, while gateway GSNs (GGSNs) act as gateways between the GPRS network and other packet data networks (PDNs). Because GPRS operates only with packet data networks, all data transmitted between mobile stations and either other mobile stations or the external network is in the form of protocol data units (PDUs). Within the GPRS network, the GPRS Tunnelling Protocol (GTP) is used for transmitting PDUs between GSNs, and this protocol is explained in some detail in the next section (see also the GTP standard [gtp97]).

4.2 The Tunnelling Protocol

A GPRS subscription involves one or more packet data protocol (PDP) addresses, each described by a PDP context in the mobile station, the SGSN and the GGSN. The PDP contexts maintain a state that indicates whether they are activated for data transfer. When data transfer between the SGSN and the GGSN is required, the network must activate the PDP contexts in both GSNs and determine a path along which data will be transmitted. This process establishes a *tunnel* used for communication. If a tunnel is not established, no data transfer can occur. If an entity on the external network attempts to send PDUs to a mobile station PDP address for which the PDP context is inactive, the PDU is discarded and an error notification is returned.

Any given tunnel may be used only for transmitting PDUs of a single network layer protocol. The two network layer protocols currently supported are X.25 and IP, though others may be added at a later date. While being transmitted over the GPRS network, PDUs of these two protocols are encapsulated with GTP headers. This encapsulation process hides the actual network layer protocol from the network, simplifying the network entities and allowing for new protocols to be more easily accommodated. The path used for transmitting these encapsulated PDUs depends on whether the network layer protocols expect a reliable link. The two network layer protocols supported, X.25 and IP, differ in their requirements for link reliability.

- The X.25 protocol relies on a reliable data link, so the GTP encapsulated PDUs are transmitted using TCP/IP. TCP, or the Transmission Control Protocol (see Postel [Pos81b]), provides facilities for retransmitting lost PDUs, thus yielding a reliable link.

- If the GTP layer is encapsulating IP (Internet Protocol, see Postel [Pos81a]) PDUs, a reliable data link is not required, and the encapsulated PDUs are transmitted using UDP/IP. UDP, or the User Datagram Protocol (see Postel [Pos80]), does not keep track of lost PDUs and thus consumes fewer network resources.

The protocol stack of the transmission plane of GTP can be seen on the right-hand side of figure 4.2, illustrating that GTP stands between the supported network layer

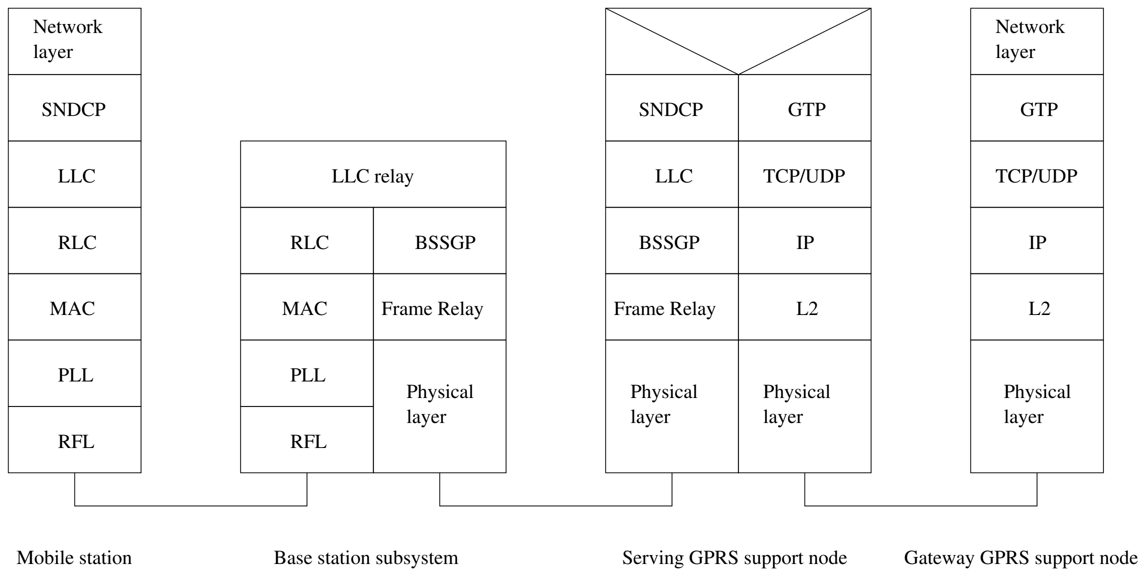


Figure 4.2: The GPRS transmission plane

protocol (X.25 or IP) and the TCP/IP or UDP/IP transmission over the GTP backbone.

So that tunnels may be activated and deactivated, GTP defines tunnel control and management messages. These messages are transmitted using UDP/IP, an unreliable communication protocol, and so this signalling aspect of GTP must arrange for its own retransmission of lost messages. A tunnel may be established in response to a request from a mobile station or, optionally, in response to the reception at a GGSN of a PDU intended for a mobile station for which an active tunnel does not currently exist. Only the former case of PDP Context Activation is considered here. Once the tunnel is established, data transfer may occur until the tunnel is destroyed during the PDP Context Deactivation process. The sequences of messages involved in PDP Context Activation and Deactivation are illustrated in figures 4.3 and 4.4. Other messages are specified for changing the quality of service required for the data link, for example.

The process of establishing a tunnel is illustrated in figure 4.5. The SGSN PDP context starts in the *inactive* state. If an Activate PDP Context Request is received from a mobile station, the SGSN creates a Create PDP Context Request and assigns to it the current Transaction ID. The Transaction ID is now incremented, ready to be assigned to the next message. The Create request is sent to the appropriate GGSN, and the SGSN PDP Context moves to the *activating* state (arc **A** in figure 4.5). If a response is not received from the GGSN within a specified time period (referred to as T3-RESPONSE), the request is resent. This process is repeated N3-REQUESTS

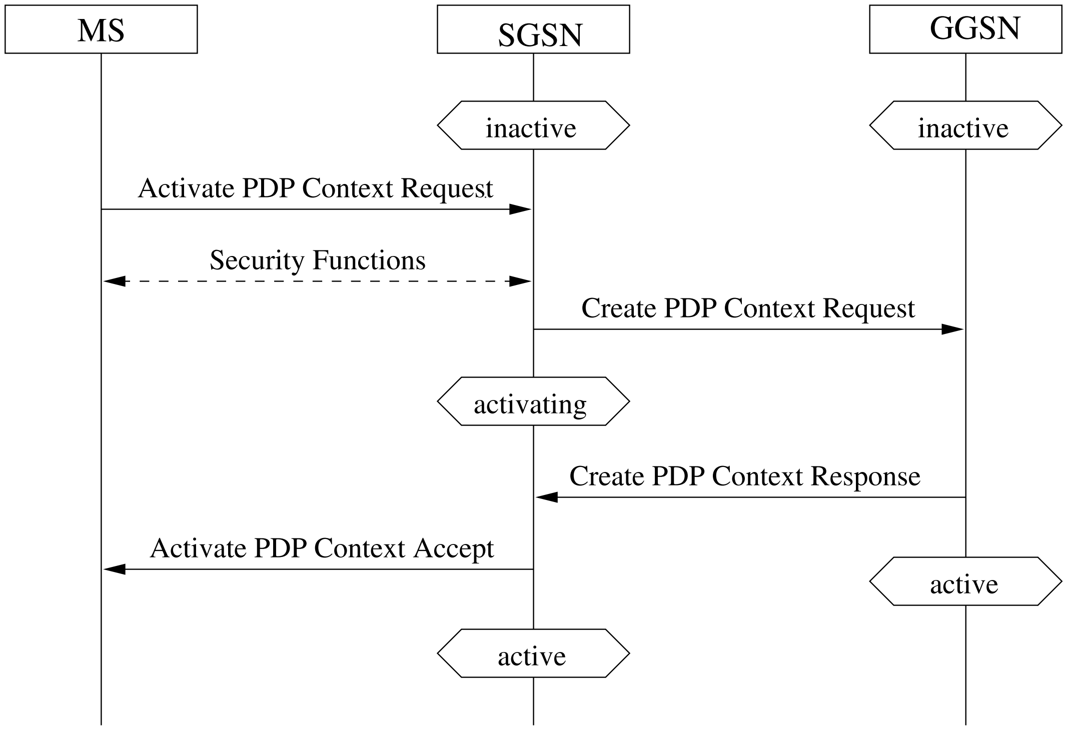


Figure 4.3: PDP Context Activation procedure

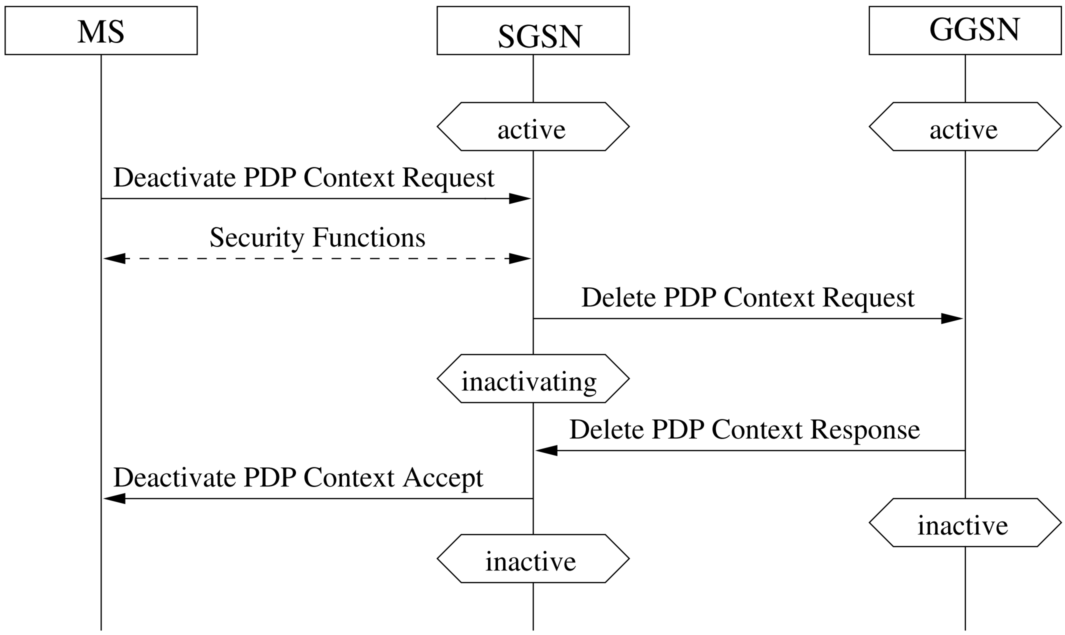


Figure 4.4: PDP Context Deactivation procedures

times, after which the SGSN returns an error status to the mobile station to indicate that a connection cannot be established and returns to the *inactive* state (arc **B** in figure 4.5). All of the retransmissions of a given request have the same Transaction ID, so the GGSN is able to recognise them as retransmissions. However, the GGSN responds to all received requests, and the Transaction ID is included in the response. For the SGSN's part, a response for which there is not an outstanding request is recognised as a duplicate and is discarded.

If a response to a Create request is received, the GGSN PDP Context is known to be active, and the SGSN PDP Context moves to the *active* state (arc **C** in figure 4.5). In this state, data transfer may occur. When the mobile station has completed all required data transfer, it may issue a Deactivate request. Alternatively, the mobile station may wish to detach from the GPRS network. Either of these events results in the SGSN PDP Context sending a Delete PDP Context Request message to the GGSN and moving to the *inactivating* state (arc **D** in figure 4.5). Again, this message may be retransmitted if a response is not received within the specified time period, and this retransmission may occur several times. If a response to the Delete request is received, or the maximum number of requests has been sent, the SGSN PDP context returns to the *inactive* state (arc **E** in figure 4.5).

Certain properties of the tunnel between GSNs, such as the Quality of Service (QoS), may be modified while the tunnel is active. If a Modify PDP Context Request is received from the mobile station, the SGSN sends an Update PDP Context Request to the GGSN and moves to the *updating* state (arc **F** in figure 4.5). As with other messages, the Update request may be retransmitted if a response is not received within T3-RESPONSE. If a response is received, the SGSN PDP Context returns to the *active* state (arc **G** in figure 4.5). However, if the maximum number of requests has been sent, it is assumed that the GGSN has failed, and the SGSN moves to the *inactive* state (arc **H** in figure 4.5).

A GPRS network will be composed of many SGSNs, each associated with a given geographical area, together with a number of GGSNs, at least one for each PDN to which the GPRS network is to be connected. Decisions about routing among these GSNs are made by reference to databases. An SGSN which has a PDU to pass to an external network must determine which GGSN is appropriate to use as a gateway, while a GGSN with a PDU to pass to a subscribed mobile station must determine which SGSN is responsible for the routing area currently occupied by the mobile user. While the route to the appropriate GGSN can be hard-coded based on the PDU type, routing to an SGSN must be based on locality information held in the Home Location Register.

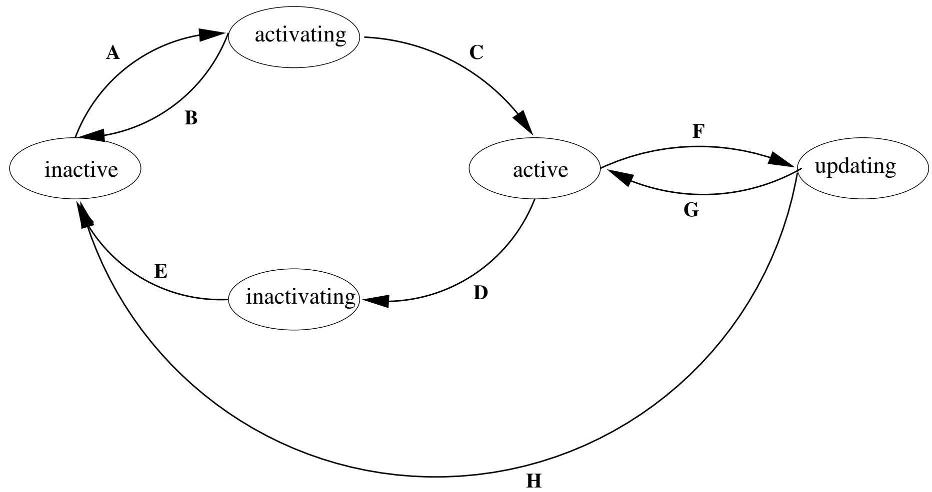


Figure 4.5: Simplified state diagram of an SGSN PDP Context. Self-loops related to timeouts (retransmission of messages) and ignored messages are not shown for clarity.

4.3 System Requirements

So that a specification could be completed in a reasonable amount of time, a simplified form of the protocol was considered for illustration here. The first simplification was to consider only one PDP context. In a real GTP system, each physical GSN would contain a large number of PDP contexts, potentially one for each PDN subscribed to by each mobile user. The combination of routing between GSNs, together with the number of PDP contexts within a single GSN, would have resulted in a prohibitively complex specification. In order to focus on the process of tunnel establishment and tear down, the system considered here was composed of one SGSN PDP context and one GGSN PDP context connected by a backbone link. In order to verify that the PDP contexts could accommodate lost messages, the link was specified in such a way that it could lose messages.

The SGSN PDP context needed to be able to receive signals indicating that the link was to be activated, and to carry out all the necessary signalling. It needed also to receive PDUs to be sent to the associated GGSN, and, assuming that the link was active, to pass these PDUs across the backbone. Similarly, assuming that the link was active, PDUs received by the GGSN from the external network needed to be passed to the SGSN and on to the mobile user.

4.4 Formalisation of Requirements

The formalisation of the system requirements was based upon a recent draft of the standard of the tunnelling protocol that indicated the possible states for each PDP context, the events which would trigger state transitions, and the actions associated with those transitions. From this standard, sequences of actions were written indicating the observable behaviour of the PDP contexts in the SGSN and GGSN. Furthermore, the standard specified the sequences of actions that should occur on the link between GSNs, in response to external stimuli. Conformance to these sequences ensures that the GSNs can operate with GSNs produced by other manufacturers.

These formalised requirements formed the basis of validation of the specification before a ROOM implementation was created. In order to perform this validation, the requirements were written either in a LOTOS format compatible with that used in the specification, or in a temporal logic format that could be used by the LMC model checker. The LOTOS format was used for sequences of expected actions, while temporal logic was used to test for properties that could be expressed in terms of orderings of events (see section 3.7 for a discussion of the use of these two techniques). Section 4.6 discusses the formalised requirements developed at this stage and their use in the validation of the specification.

4.5 Development of Specification

In order that the validation of the specification of the GTP system could be completed relatively quickly, a number of simplifications were required. In particular, the messages defined in the GTP standard consist of a sixteen octet header, followed by information that varies depending on the signal transmitted (see [gtp97] for details of the standard). Specifying this data structure in LOTOS would have greatly slowed the specification process and put undue strain on the validation tools used. Instead, it was decided that the specification should be written in terms of enumerated types representing the messages. This enumeration allowed the specification to be written more quickly and validation to be easier and more comprehensible. The resulting ROOM implementation was also written in terms of enumerated types, but can be converted to use the more complex data types to create a true implementation.

The development of the specification followed the state machines indicated in the GTP standard, and its general structure is illustrated in figure 4.6. The system was composed of two processes (`Simple_SGSN` and `Simple_GGSN`) connected by a channel process, `GTP_Backbone`. Each GSN contained a single PDP Context. The `SGSNPDPContext` was composed of two processes. The first, `SGSNPDPContext_stateMachine`, represented the signalling behaviour of the PDP context, while `SGSN_`

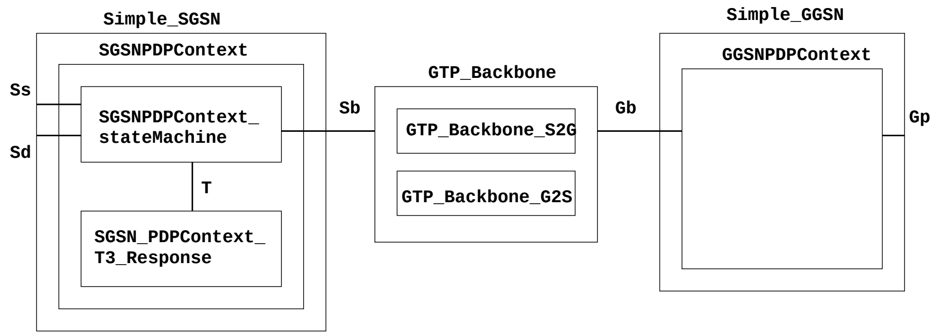


Figure 4.6: Block diagram of the GTP specification.

PDPContext_T3_Response was responsible for timing. These two processes communicated over a hidden gate, T. The SGSN PDP Context communicated with the mobile station through two gates, Ss and Sd. Respectively, these gates carried signalling messages and data. The SGSN PDP Context communicated with the GGSN through the channel, by means of gate Sb. The channel was composed of two interleaved processes, GTP_Backbone_S2G and GTP_Backbone_G2S, responsible respectively for messages from the SGSN to the GGSN, and for messages from the GGSN to the SGSN. The GGSN PDP Context was not decomposed further, and communicated with the SGSN through gate Gb and passed data to and from the external network through gate Gp.

Because the simplified SGSN contained a single PDP context, it was written as:

```
process Simple_SGSN[Ss,Sd,Sb]:noexit:=
    SGSNPDPContext[Ss,Sd,Sb]
endproc
```

The SGSNPDPContext was composed of an actor representing the state of the PDP context, together with an actor providing timing services. The use of the hide operator made the interaction between the state machine and the timer invisible outside of the SGSNPDPContext, while the parallel composition operator, |[T]|, provided a channel over which the state machine and the timer could exchange messages.

```
process SGSNPDPContext[Ss,Sd,Sb]:noexit:=
    hide T in
    (
        SGSNPDPContext_stateMachine[Ss,Sd,Sb,T]
    |[T]|
        SGSN_PDPContext_T3_Response[T]
    )
endproc
```

```
process SGSNPDPContext_stateMachine[Ss,Sd,Sb,T]:noexit:=
    SGSNPDPContext_inactive[Ss,Sd,Sb,T]
endproc
```

The initial state of the PDP context was `inactive`, as indicated by the process invocation above. Each of the five states of the state machine was then specified as a LOTOS process, with possible transitions to other states indicated as below:

```
process SGSNPDPContext_inactive[Ss,Sd,Sb,T]:noexit:=
    (
        (
        Ss !  GT_PDP_ACTIVATE_request;
        Sb !  Create_PDP_Context_Request;
        T ! timer_start;
        SGSNPDPContext_activating[Ss,Sd,Sb,T]
        )
    []
        (
        Ss !  GT_PDP_MODIFY_request;
        SGSNPDPContext_inactive[Ss,Sd,Sb,T]
        )
    []
        (
...and so on...
```

In the specification fragment above, two transitions from the state `SGSNPDPContext_inactive` can be seen. The first transition (representing arc **A** in figure 4.5), triggered by a `GT_PDP_ACTIVATE_request` from the mobile station through gate `Ss`, caused the SGSN PDP context to send a `Create_PDP_Context_Request` to the GGSN along the backbone (represented by gate `Sb`). The SGSN started a timer so that it could re-transmit the message if the GGSN did not respond within a given period, and then moved to the state `SGSNPDPContext_activating`. The second transition (a self-loop not shown in figure 4.5) was triggered by the reception of a `GT_PDP_MODIFY_request` from the mobile station and resulted simply in a self-transition to the inactive state, indicating that this message was ignored.

The timer portion of the SGSN PDP context was modelled as two actors, one of which (`SGSN_PDPContext_T3_Response_timer`) provided a simple timer capable of being started, being stopped, and sending a `timeout` signal to indicate that the timer period was over. The second actor, `SGSN_PDPContext_T3_Response_Controller`, was responsible for counting the number of timeout signals sent to the PDP context state machine, and sending a special signal (`n3_requests`) to indicate that too many timeouts had occurred (because either the GGSN or the link had failed). If the GGSN did not respond to a series of `Create_PDP_Context_Request` messages, the SGSN sent a message to the mobile user indicating that its attempts to create a tunnel had failed and returned to the inactive state.

```
process SGSN_PDPContext_T3_Response[T]:noexit:=
    hide TC in
    (
        SGSN_PDPContext_T3_Response_Controller[T, TC]
    |[TC]|
        SGSN_PDPContext_T3_Response_timer[TC]
    )
endproc
```

The specification of the GGSN PDP context was much simpler because the context occupies only one of two states (*inactive* and *active*), and there was no need to retransmit signals and so no need for a timer.

Finally, the backbone link between the SGSN and the GGSN was specified as two LOTOS processes representing the two transmission directions:

```
process GTP_Backbone[Gb, Sb]:noexit:=
    GTP_Backbone_G2S[Gb, Sb]
    |||
    GTP_Backbone_S2G[Gb, Sb]
endproc
```

The specification of the backbone was complicated somewhat by the requirement to lose messages, in order to test the retransmission procedure. As described in section 3.6.5, the channels were specified so that an arbitrary or bounded number of messages could be lost. Loss of an arbitrary number of messages was specified as:

```
process GTP_Backbone_S2G[Gb, Sb]:noexit:=
    Sb ?  x:Sb_Gb_event;
    (
        Gb !  x; GTP_Backbone_S2G[Gb, Sb]
    []
        i; GTP_Backbone_S2G[Gb, Sb]
    )
endproc

process GTP_Backbone_G2S[Gb, Sb]:noexit:=
    ...similar to above...
endproc
```

In this channel specification, the channel received a message from the SGSN and then either synchronised with the GGSN to pass on the message or lost the message and was then ready to synchronise with the SGSN again. Loss of a bounded number of messages was specified as:

```
process GTP_Backbone_S2G[Gb, Sb]:noexit:=
    Sb ? x:Sb_Gb_event;
    (
        Gb ! x;
        GTP_Backbone_S2G[Gb, Sb]
    []
        Sb ? x2:Sb_Gb_event;
        Gb ! x2;
        GTP_Backbone_S2G[Gb, Sb]
    )
endproc

process GTP_Backbone_G2S[Gb, Sb]:noexit:=
    ...similar to above...
endproc
```

In this example of the link from the SGSN to the GGSN, the link could have synchronised with the SGSN and then either synchronised with the GGSN (thus passing on the message) or received another message from the SGSN (thus discarding the first message) and passed this second message on to the GGSN. By this means, the loss of one message at a time passing between the two GSNs could be modelled. As with the alternating bit protocol (see section 3.9.2), both forms of unreliable channel specification were used during the validation procedure. The final specification contained a little over 700 lines of LOTOS, not including test sequences.

4.6 Validation and Verification

The validation and verification of the specification of the GTP system was carried out using the two techniques discussed earlier. First, sequences of LOTOS actions were composed in parallel with the specification to ensure that the observable behaviour of the system was as intended. Then, temporal logic model checking was performed to ensure that the internal behaviour of the system conformed to expectations.

4.6.1 Agent Views and Agent Scenarios

As with the alternating bit protocol, sequences of expected observed behaviour were created during the early stages of design, then were used to validate the specification. In order to check these sequences against the specification, a state space exploration tool was used. Early trials using the SELA tool indicated that the exploration of the composition of each scenario with the specification could not be carried out to a sufficient depth within the memory available on the machine used (a Sun Ultra 1 with 96MB of memory). Instead, the LOLA tool (described on page 21) was used, because it makes more efficient use of memory. LOLA composes the test sequence in parallel with the specified system and explores the state space.

As explained on page 102, the GTP specification was written with two different channel specifications: one which would lose fewer than N3_REQUESTS messages, and one which could lose an arbitrary number of messages. The two channel specifications required that different validation sequences be used:

1. If the channel would lose fewer than N3_REQUESTS messages, the retransmission feature of GTP ensures that a message sent from the SGSN will eventually arrive at the GGSN. Thus, the validation sequences were written such that an attempt to establish a tunnel *would* succeed, for example.

2. If the channel could lose an arbitrary number of messages, a message from the SGSN cannot be assumed to arrive at the GGSN, so the SGSN may report failure to the mobile station.

Given these differences, each scenario was represented by two different validation sequences. The first, used with a bounded loss channel, checked that the SGSN did try more than once to establish a tunnel. The second sequence assumed a more realistic channel that could lose an unbounded number of messages, but had to be written to accommodate the possible failure of the system to establish a tunnel. These two sequences are illustrated by the example below.

The first example scenario modelled the system activating the link between the SGSN and the GGSN and then sending a PDU from the mobile station to the external network. The corresponding validation sequence, assuming bounded message loss, was:

```
process must_pass_1[Ss, Sd, Gp, success, failure]:noexit:=
    (* MUST PASS 1:  Check that we can activate the link and *)
    (* send a pdu from the MS to the network.  *)
    Ss !  GT_PDP_ACTIVATE_request;
    Ss !  GT_PDP_ACTIVATE_confirm_SUCCESSFUL;
    Sd !  GT_UNITDATA_indication_send;
    Gp !  recvd_PDU;
    success;
    stop
endproc
```

Because message loss was bounded, the sequence could be written assuming that the SGSN would respond to a `GT_PDP_ACTIVATE_request` with a `GT_PDP_ACTIVATE_confirm_SUCCESSFUL`. Having established a channel, the validation sequence models the mobile station sending a PDU (`GT_UNITDATA_indication_send`) to be received at the other end (`recvd_PDU`). If, on the other hand, the channel was specified so that it could lose an unbounded number of messages, the validation sequence had to be written on the assumption that a message could be lost more than N3_REQUESTS times, so that the SGSN would report failure:

```
process must_pass_1[Ss, Sd, Gp, success, failure]:noexit:=
    (* MUST PASS 1:  Check that we can activate the link and *)
    (* send a pdu from the MS to the network.  *)
    Ss !  GT_PDP_ACTIVATE_request;
    (
        (
            Ss !  GT_PDP_ACTIVATE_confirm_FAILURE;
            success;
            stop
        )
    []
        (
            Ss !  GT_PDP_ACTIVATE_confirm_SUCCESSFUL;
            Sd !  GT_UNITDATA_indication_send;
            Gp !  recvd_PDU;
            success;
            stop
        )
    )
endproc
```

When these validation sequences were composed in parallel with the corresponding versions of the specification, the compositions should have executed the actions in the order indicated, eventually reaching the special event `success`. In the first case, reaching `success` would indicate that the link was successfully established. In the second case, however, reaching `success` may indicate either that the channel was established and a PDU sent, or that the channel establishment failed. At this level of validation, either of these is a correct response, though of course the `GT_PDP_ACTIVATE_confirm_FAILURE` result should be seen only if the SGSN did not receive a response from the GGSN in the required time. Testing for this relationship between the response from the GGSN and the `FAILURE` result during the temporal logic model checking is described in section 4.6.2. Using LOLA with the test sequences above produced *must pass* results, indicating that the LOTOS specification of the GTP system conformed to this formalised requirement, allowing for establishment of a tunnel and the transmission of a PDU.

Similar sequences of actions were created to represent other desired scenarios. The following sequence models the activation of a link, followed by the transmission of a PDU from the network to the mobile station, followed by deactivation of the link:

```
process must_pass_2[Ss, Sd, Gp, success, failure]:noexit:=
    (* MUST PASS 2:  Check that we can activate the link, *)
    (* send a pdu from the network to the MS and then *)
    (* deactivate the link *)
        Ss !  GT_PDP_ACTIVATE_request;
    (
        (
            Ss !  GT_PDP_ACTIVATE_confirm_FAILURE;
            success;
            stop
        )
    []
        (
            Ss !  GT_PDP_ACTIVATE_confirm_SUCCESSFUL;
            Gp !  send_PDU;
            (
                Sd !  GT_UNITDATA_indication_recvd;
                Ss !  GT_PDP_DEACTIVATE_request;
                Ss !  GT_PDP_DEACTIVATE_confirm;
                success;
                stop
            []
```

```
                    Ss ! GT_PDP_DEACTIVATE_request;
                    Ss ! GT_PDP_DEACTIVATE_confirm;
                    success;
                    stop
                )
            )
        )
    endproc
```

When this sequence was composed in parallel with the GTP specification with a channel allowing unbounded loss, and expanded to a depth of 20 transitions[3], the LOLA tool reported a *may pass* result, indicating that 65 successes were reached, but that one expansion was truncated by the depth limit. Given more time, the LOLA tool would be able to expand the test further to report on the longer expansion.

Furthermore, sequences were created representing undesirable scenarios, such as sending a PDU before the link is activated, to check whether the specification prevented erroneous behaviour. The example below tested whether the GTP specification allowed the mobile station to modify the link before the link was established:

```
process must_fail_1[Ss, Sd, Gp, success, failure]:noexit:=
    (* MUST FAIL 1: Check that we can't send a modify request *)
    (* without having previously activated the link *)
    Ss ! GT_PDP_MODIFY_request;
    Ss ! GT_PDP_MODIFY_confirm_SUCCESSFUL;
    failure;
    stop
endproc
```

When this test was composed in parallel with the GTP specification, LOLA quickly reported that the combination deadlocked, indicating that the system would not accept this inappropriate command sequence.

4.6.2 Model Checking

In this GTP case study, model checking was used to verify relationships between externally visible events and events occurring on the SGSN-GGSN backbone connection. Although temporal logic formulae would be tested in an ideal case against the GTP specification alone, in practice the state space explosion of the specification surpasses reasonable time and memory constraints. Instead, the system was composed with

[3]The LOLA tool allows the user to limit the time taken for the test by specifying the depth to which expansion should be carried out.

test sequences in order to constrain the state space expansion. The test sequences consisted of sequences of observable actions to trigger interactions between the SGSN and the GGSN. Temporal logic requirements on the relationship between observable and internal events were then specified and checked using the LMC model checker. The requirements checked are described below.

Following the Response Property Pattern of Dwyer et al. (see section 3.5.3) with *global* scope, the requirement that the SGSN should follow the reception of a `GT_PDP_ACTIVATE_request` on the `Ss` port with the transmission of a `Create_PDP_Context_Request` along the backbone was represented as:

$$AG((Ss\,!\,GT\_PDP\_ACTIVATE\_request) \rightarrow \\ AF((Sb\,!\,Create\_PDP\_Context\_Request)))$$

For this example, the test sequence `must_pass_1`, seen on page 104, was used to restrain the state space expansion. This formula was checked against the restricted state expansion using the LMC model checker which indicated that the formula holds.

As indicated earlier, the observed response of the SGSN PDP context to a request to activate the link may indicate either the success or failure of this request. However, the SGSN should indicate failure only if it has tried a number of times to transmit the request to the GGSN. This behaviour of the system cannot be verified by simply observing the sequence of events at the `Ss` gate, but by monitoring the signals from the internal timer, confidence in the correct functioning of the SGSN PDP context can be gained. Before verification, however, it was necessary to modify the specification from its original form to expose the communication channel between the state machine of the SGSN PDP context and the timer, as seen below. This modification was required because the LMC model checker cannot check for the presence of hidden actions.

```
process SGSNPDPContext[Ss,Sd,Sb,T]:noexit:=
(* process SGSNPDPContext[Ss,Sd,Sb]:noexit:= *)
    (* hide T in *)
    (
        SGSNPDPContext_stateMachine[Ss,Sd,Sb,T]
    |[T]|
        SGSN_PDPContext_T3_Response[T]
    )
endproc
```

With the `hide` operator commented out, and the timer channel added to the gate list for the `SGSNPDPContext`, it was then possible to use model checking to verify that the state machine issued a failure message only if the timer indicated that `n3_requests` had occurred. This condition was expressed, following the Precedence Property Pattern of Dwyer et al. with *global* scope, as:

$$A(\neg(Ss\,!\,GT\_PDP\_ACTIVATE\_confirm\_FAILURE)\,\mathcal{U}\,((T\,!\,n3\_requests) \vee AG(\neg(Ss\,!\,GT\_PDP\_ACTIVATE\_confirm\_FAILURE))))$$

Again, to constrain the state space expansion of the specification, the formula was verified against the specification composed in parallel with the `must_pass_1` test sequence. The above formula was verified using LMC.

As indicated above, the temporal logic model checking was used largely in this case study to verify the correct relationship between observable events and internal events. For example, once a PDP context is established, reception of a PDU by the GGSN from the external network should result in the GGSN attempting to transmit that PDU to the SGSN. Using Dwyer's Response Property Pattern with *after* scope, this requirement was stated as:

$$AG(Ss\,!\,GT\_PDP\_ACTIVATE\_confirm\_SUCCESSFUL \longrightarrow AG(Gp\,!\,send\_PDU \longrightarrow AF(Gb\,!\,G\_PDU)))$$

This formula was demonstrated to hold to the specified expansion depth by the LMC tool.

The last formula checked is based on a different test sequence. The sequence models the mobile station activating the link, deactivating the link and then, before confirmation of the deactivation is received, attempting to activate the link again. The GTP standard indicates that, in this circumstance, the SGSN should not send a `Create_PDP_Context_Request` to the GGSN until the deactivation has been confirmed. The test sequence used was:

```
process test_sequence[Ss, Sd, Gp, success, failure]:noexit:=
    (* activate, deactivate, then activate again *)
    (* before the confirm *)
    Ss !  GT_PDP_ACTIVATE_request;
    (
        (
            Ss !  GT_PDP_ACTIVATE_confirm_FAILURE;
            stop )
    []
        (
            Ss !  GT_PDP_ACTIVATE_confirm_SUCCESSFUL;
```

```
                Ss !  GT_PDP_DEACTIVATE_request;
                Ss !  GT_PDP_ACTIVATE_request;
                Ss !  GT_PDP_DEACTIVATE_confirm;
                stop )
        )
    endproc
```

The requirement that a given event (the `Create_PDP_Context_Request` not occur is an example of Dwyer's Absence Property Pattern. The temporal scope of the requirement is *between*, resulting in the following formula:

$$\begin{aligned} AG(Ss\,!\,GT\_PDP\_DEACTIVATE\_request \rightarrow \\ A(\neg Sb\,!\,Create\_PDP\_Context\_Request\,\mathcal{U} \\ (Ss\,!\,GT\_PDP\_DEACTIVATE\_confirm\ \vee \\ AG(\neg Ss\,!\,GT\_PDP\_DEACTIVATE\_confirm)))) \end{aligned}$$

The LMC model checker reported that this formula held on the system produced by composing the specification in parallel with the above test sequence.

This section has illustrated some of the agent scenario sequences and temporal logic formulae used in validation of the LOTOS specification. It should be noted that the state machine for the SGSN PDP Context involved five states with approximately ten possible transitions per state, according to the event received by the PDP Context. As such, the coverage provided by the sequences and formulae in this section is relatively small — in a real development environment, much more validation and verification would be required before moving on to implementation.

4.7 Implementation of the Specification in ROOM

As with the alternating bit protocol example, the structure of the LOTOS specification mapped onto the state-machine based ROOM notation relatively easily. The top-level structure of the system, composed of the SGSN and GGSN linked by the backbone, translated into an actor containing actors representing the SGSN and GGSN, as in figure 4.7. Note that both actors have a small symbol in the lower left-hand corner, indicating that they both contain other constituent actors. The gates `Sb` and `Gb` appear in this diagram as `backboneR1`.

Having specified the structure of the GTP system in the ROOM notation, it was then necessary to specify its behaviour. For example, the state machine representing the SGSN PDP context could be in one of five states, resulting in the ROOMchart

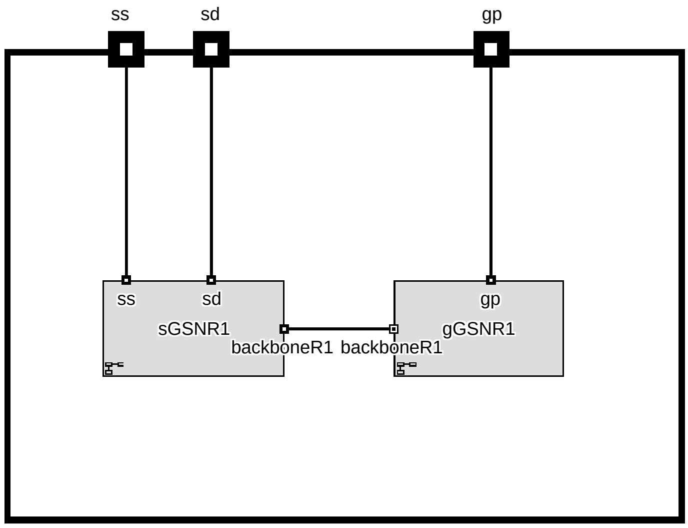


Figure 4.7: Structure of the GTP system in the ROOM notation

seen in figure 4.8. The state machine includes additional transitions not shown in figure 4.5, occurring when the timer times out, for example.

Each of the transitions in the model represented one of the cases within the LOTOS description of a state. For example, in the specification fragment of the `inactive` state seen above, there appeared the following sequence of LOTOS actions:

```
(
Ss !  GT_PDP_ACTIVATE_request;
Sb !  Create_PDP_Context_Request;
T ! timer_start;
SGSNPDPContext_activating[Ss,Sd,Sb,T]
)
```

This sequence indicated that if the SGSN PDP context received a `GT_PDP_ACTIVATE_request`, it should send a `Create_PDP_Context_Request` message, start the timer, and move to the `activating` state. This fragment appeared in the ROOMchart as a transition from the `inactive` state to the `activating` state, triggered by the appropriate message, with action code to send a create request to the GGSN and start the timer.

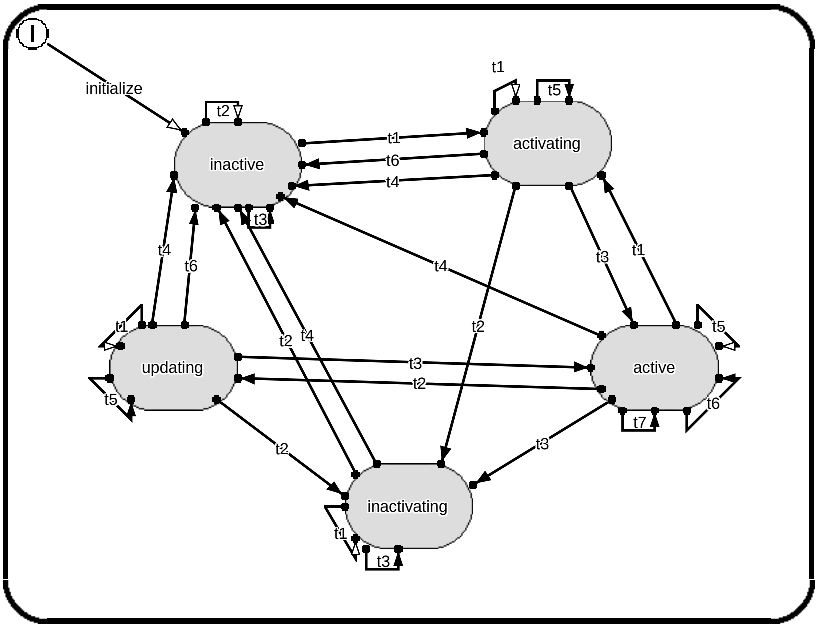


Figure 4.8: ROOMchart of the behaviour of the SGSN PDP context

4.8 Executing the ROOM Model

The ROOM model of the GTP system was compiled and executed within the Simulation RTS. As before, the ObjecTime toolset allowed *daemons* to be attached to ports of the system, thus permitting messages to be injected and trace windows to be opened to observe the flow of messages through ports. Also, the toolset provides support for creating trace windows to watch the exchange of messages between system entities. In order to validate the ROOM model, one of the sequences that was used in the validation of the LOTOS specification was manually applied to the ROOM model. In the example below, a `GT_PDP_ACTIVATE_request` was injected into the `Ss` port of the SGSN, followed by a `GT_UNITDATA_indication_send` into the `Sd` port and a `GT_PDP_DEACTIVATE_request` into the `Ss` port. The toolset simulated the behaviour of the system given these incoming messages, yielding the message sequence chart seen in figure 4.9.[4]

[4]Unfortunately, the MSC generated by the toolset does not include the initial `GT_PDP_ACTIVATE_request`.

The four vertical lines represent the four system entities whose behaviour was traced: the GTP system as a whole (as in figure 4.7), the state machine of the SGSN PDP context, the timer and the GGSN PDP context. The hexagons indicate the current state of each entity. Reading from the top of the diagram, the following sequence of events and states was observed:

1. The SGSN PDP Context sends a signal to the timer requesting that it start and then moves to the *activating* state.

2. The timer gives a *timeout* event but, before retransmission can occur, the GGSN PDP Context returns a response to the SGSN. This response causes the SGSN to stop the timer and send a message out on the `Ss` gate indicating that the link has been established. Both the SGSN and the GGSN are now in the *active* state.

3. The SGSN PDP Context receives a PDU from the mobile station and responds by forwarding the PDU to the GGSN PDP Context. The GGSN signals that the PDU has been received by a message on the `Gp` port.

4. The SGSN PDP Context receives a Deactivate request from the mobile station. It sends a `Delete_PDP_Context_Request` to the GGSN, starts the timer and moves to the *inactivating* state.

5. When the GGSN receives the request, it sends a response and moves to the *inactive* state.

6. When the SGSN PDP Context receives the response, it stops the timer, sends a message to the mobile station indicating that the link has been deactivated, and moves to the *inactive* state.

4.9 Summary

The design methodology described in the preceding chapter was applied to a significant example protocol: the GPRS Tunnelling Protocol. The description of the protocol was taken from the relevant standards document, and used to create formal requirements as both agent scenarios and temporal logic formulae. A LOTOS specification was written, using the style specified in the methodology. This specification was then validated using validation sequences derived from agent scenarios and using temporal logic model checking. The validated specification was then used to derive a ROOM model, using the mapping described in the methodology, and the ObjecTime

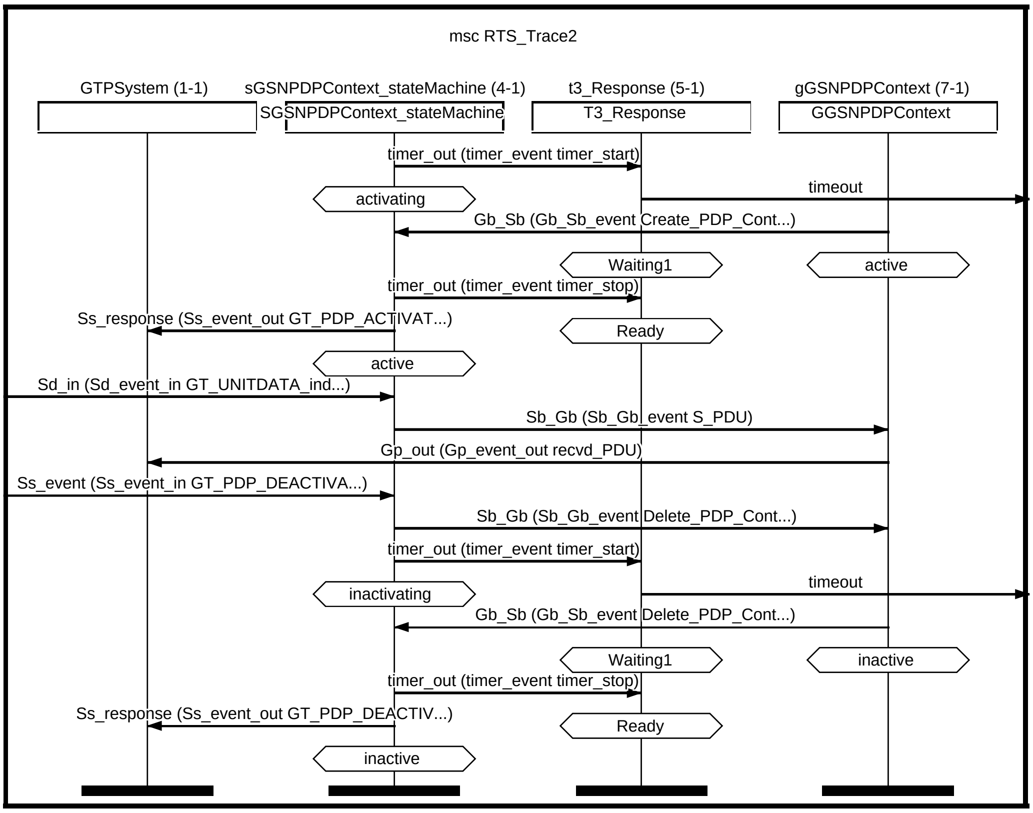


Figure 4.9: Message Sequence Chart illustrating the behaviour of the GTP system.

toolset was used to create a C++ implementation. Finally, the implementation was animated in the ObjecTime toolset to demonstrate its correct operation.

Chapter 5

The POP3 Mail Protocol

This chapter describes a simple example illustrating the discussion of the representation of inheritance in LOTOS in section 3.6.4. The example chosen was the authorisation phase of the POP3 mail protocol. The case study was much smaller than the GPRS Tunnelling Protocol, and was not subject to the same level of validation. However, the case study does provide a good example of how inheritance can ease the production of a more advanced version of a protocol, building on the work of producing the basic protocol.

5.1 The POP3 Protocol

Although the World Wide Web has received much more publicity, electronic mail (or email) remains a popular and important application of computer networks such as the Internet. Email systems at companies or universities are generally designed on the basis of machines' permanent connections to the network; mail destined for a particular user can thus be delivered directly and immediately. However, this model is not suitable for the majority of home users who have only intermittent connections to the network. Attempting to deliver mail directly would result in an enormous number of failed attempts before eventual connection. In order to address this problem, a maildrop system was devised, the latest version of which is called POP3[1] (see Myers and Rose [MR96]).

If a user handles mail using the POP3 maildrop service, all mail intended for that user will be delivered to a machine (called the *mail server*) that does have a permanent connection to the network. When the user's machine is connected to the network, the POP3 protocol allows the user's mail client to connect to the server, determine

[1] POP3 is now superseded in some mail networks by the IMAP protocol (see Crispin [Cri94]), which offers better mail manipulation services.

how many messages are available, download chosen messages and delete messages from the server. The first step in connecting to the server is known as *authorisation*, and requires users to identify themselves and prove their identity before being given access to their mail box.

The POP3 protocol allows two authorisation methods. The first, known as the USER and PASS command combination, is common but involves the security concern of sending the user's password in plain text across the network. The authorisation procedure involves an exchange of messages such as the one below:

```
+OK POP3 server ready
USER mrose
+OK
PASS secret
+OK mrose's maildrop has 2 messages (320 octets)
```

If the user-name and password combination is incorrect, the last line of the exchange above is replaced by the message `-ERR`, indicating that access to the maildrop has not been granted.

The second authorisation method involves the more secure APOP command, which does not send the user's password in the clear. If the POP3 server implements the APOP command, its greeting will include a time-stamp of the form:

```
<processID.clock@hostname>
```

where the `processID` is the decimal value of the server's process identifier, `clock` is the decimal value of the system clock, and `hostname` is the fully-qualified domain name of the server's host system. Combining these three pieces of information means that each time-stamp will be unique. The POP3 client combines this time-stamp with a secret known only to the client and the server, and applies the MD5 algorithm (see Rivest [Riv92]) to yield a 16-octet value that is returned as part of the APOP command. Meanwhile, the server carries out the same calculation and then compares its value with the client's value to determine whether access to the maildrop should be permitted. The APOP command will resemble the following:

```
APOP mrose c4c9334bac560ecc979e58001b3e22fb
```

where the string following the user's name is the hexadecimal representation of the MD5 digest.

Once authorisation has been completed, the POP3 system enters the TRANSACTION state, in which commands to retrieve and delete mail may be issued. For the purposes of this case study, however, only the authorisation process will be considered.

5.2 System Requirements and Their Formalisation

Although a true implementation of a POP3 server would provide facilities to determine the number of messages in the maildrop, to retrieve and delete messages, and so on, this chapter is concerned only with the authorisation phase of the server's operation. Initially, the server was designed to provide facilities for authorisation through the USER and PASS command combination. Section 5.4 describes how the functionality of the server was extended to allow for authorisation using the APOP command, thus illustrating how the use of inheritance allowed reuse of previous work in specifying the basic server.

The formal requirements of the authorisation phase of the POP3 server were expressed in terms of sequences of desired actions. Assuming that a correct username/password combination was supplied, the sequence of actions seen in section 5.1 should culminate in an `+OK` message. If the username/password combination was incorrect, the same sequence should culminate in `-ERR`

5.3 Development of Specification

To simplify the development of the LOTOS specification, in order that the important points about inheritance may be illustrated, enumerated data types were used. Thus, for example, the possible user names were enumerated as:

```
type user_name is
    sorts user_name
    opns mrose, frated :-> user_name
endtype
```

The top-level behaviour of the system was defined as a server composed in parallel with a tester process, used to ensure that the desired sequences of actions could be observed:

```
behaviour
(
    POP3_server[s]
    |[s]|
    POP3_tester[s, success]
)
```

The POP3 server was specified in the state-oriented style seen in previous sections:

```
process POP3_server[s]:exit:=
    s!ok!server_ready; (* send greeting to user *)
    waiting_for_user[s]
where

    process waiting_for_user[s]:exit:=
        s!user?x:user_name; (* receive USER <username> *)
        s!ok;           (* respond with +OK *)
        waiting_for_password[s](x)
    []
        s!quit; (* receive QUIT *)
        exit
    endproc

    process waiting_for_password[s](x:user_name):exit:=
        s!pass?y:pword; (* receive PASS <password> *)
        (
            [y = correct_pword]->
                s!ok; (* password correct, so respond *)
                authenticated[s] (* with +OK *)
            []
            [y = incorrect_pword]->
                s!err; (* respond with +ERR and allow *)
                waiting_for_user[s] (* user to try again *)
        )
    []
        s!quit;
        exit
    endproc

    process authenticated[s]:noexit:=
        stop
    endproc
endproc
```

5.4 Extending the Authentication Method

The specification fragment above illustrates a POP3 server that can accept only login attempts through the USER and PASS command combination. It may be desirable also to provide an enhanced server that can support both the basic authorisation method and the APOP method, so that more advanced clients can be supported. This enhancement can be performed by copying the existing code and adding new code to support the APOP method. However, this copying would yield two separate specifications and would hide the fact that one is really an enhancement of the other. Inheritance offers a way of taking some of the behaviour from the basic authorisation system and adding new behaviour, while making it clear that the new system still relies upon the basic system.

The inheritance of the enhanced server may be indicated using the syntax described in section 3.6.4:

```
process POP3_server_enhanced[s]:exit:=
extends process POP3_server[s]:exit:=
    s!ok!timestamp;
    waiting_for_user_enhanced[s]
where
    process waiting_for_user_enhanced[s]:exit:=
    extends process waiting_for_user[s]:exit:=
        s!apop?x:user_name?y:apop_value;
        (
            [y = correct_apop_value] ->
                s!ok;
                authenticated[s]
            []
            [y = incorrect_apop_value] ->
                s!err;
                waiting_for_user_enhanced[s]
        )
    endproc
endproc
```

The inheritance syntax specifies the way in which the enhanced server differs from the basic server. The initial greeting of the enhanced server includes a timestamp that is used by clients capable of APOP authorisation to determine the digest value. The process `waiting_for_user_enhanced` extends the process `waiting_for_user` (capable only of accepting the commands `USER` and `QUIT`) to accept the `APOP` command. The syntax above could be expanded using an automatic preprocessor to yield standard LOTOS. An automatic preprocessor has not yet been implemented, however, so this expansion was carried out manually to yield the definition of `waiting_for_user_enhanced` below:

```
process waiting_for_user_enhanced[s]:exit:=
    s!user?x:user_name;
    waiting_for_password[s](x)
[]
    s!apop?x:user_name?y:apop_v;
    (
        [y = correct_apop_value]->
            s!ok;
            authenticated[s]
        []
        [y = incorrect_apop_value]->
            s!err;
            waiting_for_user_enhanced[s]
    )
[]
    s!quit;
    exit
endproc
```

The final specification contained a little under 100 lines of LOTOS, not including test sequences.

5.5 Validation

As discussed in section 5.2, simple validation of the server specification was carried out by composing it in parallel with an expected sequence of actions to determine whether the specification supported that sequence. A carefully written sequence made it possible to demonstrate that the enhanced server supported all of the behaviour of the basic server together with its new behaviour. For example, a sequence to validate an ordinary authorisation procedure was specified as:

```
process POP3_tester[s, success]:noexit:=
    s!ok?x:server_msg_value;
    s!user!mrose;
    s!ok;
    s!pass!correct_pword;
    s!ok;
    success;
    stop
endproc
```

Allowing the first line to synchronise with any `+OK` message ensured that the validation sequence could work with both the ordinary POP3 server, which greets the user with:

```
+OK POP3 server ready
```

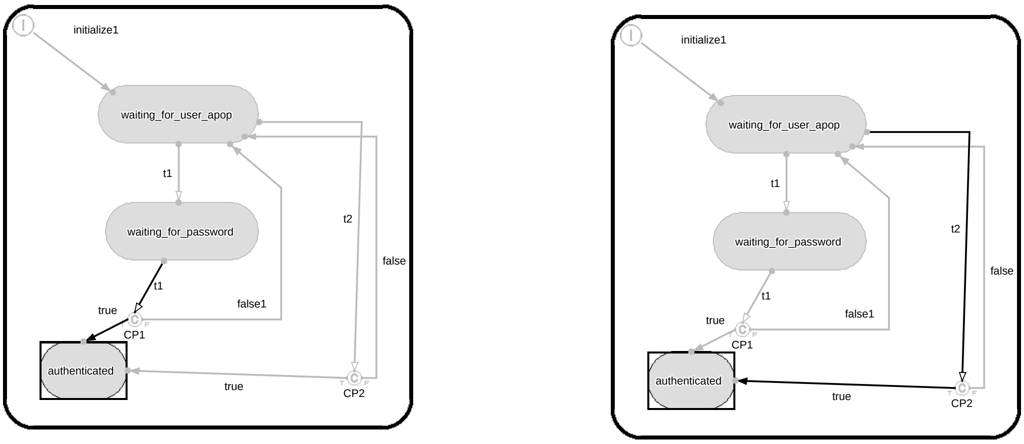
and the enhanced POP3 server, which greets the user with:

```
+OK POP3 server ready <process-ID.clock@hostname>
```

When this sequence was composed in parallel with either the basic server or the enhanced server, and was executed using LOLA, the `success` action was reached, thus indicating that both servers supported the basic authorisation procedure. Further validation sequences could be constructed to test whether the server refuses invalid passwords or to provide further robustness testing.

5.6 Implementation of the Specification

Given the extended and validated specification, both the basic and enhanced POP3 server authorisation phases were implemented using the inheritance mechanism of the ROOM notation. First, an actor class representing the basic server was constructed, with a port representing its connection with the client and states corresponding to the processes `waiting_for_user`, `waiting_for_password` and `authenticated`. A subclass was then created to represent the enhanced server, inheriting the `waiting_for_password` and `authenticated` states from the basic server, but replacing the `waiting_for_user` state with `waiting_for_user_enhanced`, and with added transitions to check the value of the APOP digest. The model was compiled and executed using the ObjecTime toolset. As discussed in section 3.9.6, the sequences used in the validation of the LOTOS specification were used to manually validate the ROOM model to ensure that the server allowed for both USER/PASS authorisation and APOP authorisation. Figure 5.1 illustrates the execution of these two sequences,



(a) Authorisation through USER and PASS commands

(b) Authorisation through APOP command

Figure 5.1: ObjecTime execution traces illustrating the two forms of authorisation supported by the enhanced POP3 server.

demonstrating that it was possible to reach the `authorised` state through either authorisation sequence. Figure 5.1(a) illustrates the state machine for the enhanced server after a USER and PASS command authorisation, while figure 5.1(b) illustrates the state machine after an APOP authorisation.

5.7 Summary

To illustrate the discussion of inheritance in section 3.6.4, the methodology was applied to the POP3 mail protocol. First, a specification of a server that would support the USER and PASS command authorisation was written. It was shown that this specification could be extended to support APOP authorisation and, further, that this extension could be performed using inheritance. The use of inheritance allowed the specification of the new server to be written by specifying only the new behaviour; the support for the USER and PASS command authorisation was inherited from the basic server.

The inheritance mechanism relies upon a preprocessor to expand certain constructs in a LOTOS specification. An automatic preprocessor does not yet exist, and so the expansion was performed manually. A ROOM model of the two server specifications was then derived and executed to demonstrate the correct operation of the servers.

Chapter 6

Conclusions and Further Work

6.1 Summary

This thesis presents a protocol validation and implementation methodology based on the combined use of the LOTOS formal description technique and the ROOM notation. LOTOS was shown to be a language suitable not only for abstract specification, but also for specifying object-oriented implementations of systems. Using LOTOS as part of the design methodology made available the techniques that have been developed for validation of LOTOS specifications. In particular, the availability of state-space expansion tools made possible model checking that could not have been performed on ROOM models alone. Two validation techniques for specifications were described:

1. A specification to be validated was composed in parallel with execution sequences derived from a description of the system in terms of agent views. This technique was used mainly to validate specifications against initial requirements (see section 3.7.1).

2. Temporal logic formulae were written to describe both initial requirements and design features. The specification to be validated was checked against these temporal logic formulae using the LMC model checker (see section 3.7.2).

Using ROOM provided a connection between a validated LOTOS specification and a working implementation in the C++ programming language. By demonstrating that a validated specification can be mapped into a system model in the ROOM notation, confidence was gained in the correct operation of the final implementation. Furthermore, the scenarios used in the validation of the LOTOS specification were reused to manually validate the ROOM model, increasing confidence in the implementation's correct operation.

The thesis went on to illustrate the use of the methodology by reference to two industrially-relevant protocols. The first case study applied the methodology to the GPRS Tunnelling Protocol, demonstrating many of the general concepts of the methodology. The second case study, involving the authorisation procedure of the POP3 Internet mail protocol, indicated the usefulness of inheritance in developing enhancements of existing systems.

6.2 Contributions of the Thesis

The thesis makes a number of contributions, the most significant of which are listed below:

Created new design methodology The thesis describes a new design methodology that combines the LOTOS formal description technique and the ROOM notation. Appropriate subsets of the two notations are described with a mapping that facilitates the derivation of ROOM implementation models from validated LOTOS specifications. Combining the two notations in a methodology allows the designer to benefit from the facilities of both: LOTOS permits for varying levels of abstraction in the creation of a design, and allows for the application of a number of validation techniques; ROOM provides a graphical execution environment and the automatic creation of executable code.

Constructed implementation-oriented LOTOS specification style The thesis describes a LOTOS specification style that is suitable for ultimate implementation through the ROOM notation. The style provides for hierarchical decomposition of actors, allowing a distributed system to be described in terms of self-contained actors. The style describes actor behaviour in terms of hierarchical state machines and provides for synchronous or asynchronous communication between actors.

Devised expression of inheritance in LOTOS In the creation of an implementation-oriented specification style, the thesis demonstrates how important aspects of object-orientation such as encapsulation and inheritance may be expressed. The discussion of inheritance in section 3.6.4 extends the earlier work of Mayr and Rudkin to offer a powerful and succinct expression of an inheritance relation. Furthermore, the inheritance method presented in this thesis does not require changes to the basic syntax of LOTOS, allowing the designer to use existing LOTOS tools.

Illustrated validation using agent scenarios The thesis demonstrates the part that agent scenarios can play in the specification of formal system requirements and subsequent validation. Building on the work of Clark and Moreira (see section 3.5.2), the thesis illustrates how important system characteristics can be specified and validated. The agent scenarios are used not only to validate the LOTOS specification, but also to validate the resulting ROOM implementation model.

Applied temporal logic patterns to validation The thesis demonstrates the part that temporal logic can play in the validation and verification of LOTOS specifications of protocol systems. In particular, the property specification patterns of Dwyer's group are applied to LOTOS (see section 3.5.3). The application of these patterns appears to offer a more tractable means of applying the power of temporal logic to the validation and verification of LOTOS specifications.

6.3 Further Work

While the thesis has indicated that a design methodology combining LOTOS and ROOM may provide a useful way of validating protocol designs before implementation, further work is required before the methodology is applicable to industrial use. Among the challenges raised by the thesis are:

Automate mapping While the mapping between the LOTOS specification style (described in sections 3.6.2 to 3.6.5) and the ROOM notation is expressed unequivocally, the mapping has not yet been automated. The automation of the mapping would be a substantial piece of work, but would not have added greatly to the exploratory research described in this thesis. Further work could explore automatic translation which would avoid the problem of introducing errors during the manual conversion from LOTOS to ROOM. Similarly, the method for expressing inheritance introduced in section 3.6.4 would gain from the provision of an automatic translator.

Use E-LOTOS The thesis currently concerns itself only with the 1989 standard of LOTOS [fS89]. The upcoming E-LOTOS standard includes a number of enhancements to the language, such as support for quantitative time specification. Future work could consider adding quantitative time to the specification, an addition that would allow validation of quantitative, rather than just qualitative, temporal properties. Such research could also draw upon the work of Kremer, who used stochastic Petri Nets in conjunction with LOTOS specifications to examine the performance of protocols [Kre95]. Other enhancements available in E-LOTOS may also allow further development of this methodology to include type-checking on gates so that illegal messages may be detected at compile time.

Validate ROOM model The methodology uses agent scenarios derived from requirements to validate the LOTOS specification. These scenarios are then used to manually validate the ROOM model by injecting messages as indicated in the scenarios and using daemons in the ObjecTime toolset to watch message flows. A useful area of future work would be the automation of this process so that the ROOM model is directly and automatically validated against requirements. This work may require enhancements to the ObjecTime toolset. Also, as mentioned in section 1.2, this work would require a formal validation semantics for ROOM.

Model interrupt behaviour The methodology does not as yet include any consideration of interrupt behaviour. Reactive systems need to be able to react to an interrupt regardless of their current state and, when interrupt processing is complete, return to their original state. Although the ROOM notation includes group transitions and history transitions that, together, may be used to model interrupt behaviour, specifying interrupts in LOTOS is not as easy. Hernalsteen and Février [HF97] developed Stepien's suggestions for an extension to LOTOS involving a suspend/resume operator that would provide support for some form of interrupt behaviour. This extension has since been accepted as part of the E-LOTOS standard. Future work could extend the methodology to include interrupts (see section 3.6.6).

Add dynamic creation of actors All actors discussed in this thesis are static; systems have their full complement of actors from the beginning. Reactive systems often involve multiple copies of actors that can be dynamically created and deleted to meet demand. For example, server software can dynamically create a handler process for each incoming connection, deleting the handler when the connection is broken. This dynamic actor creation can be specified in LOTOS using recursive interleave (see Stepien and Logrippo [SL94] and [SL93]), and future work could extend the subset of LOTOS considered in this thesis to allow for dynamic systems. The ROOM notation includes the notion of optional replicated actors that can be created and deleted under the control of another actor, and that could provide an implementation route for dynamic systems.

Extend LOTOS data types In order to expedite the production of specifications, only enumerated data types have been considered in this thesis. Further research is needed to explore the possibility of representing real data types, bearing in mind that industrial protocols may require large data structures (for example, GTP requires a sixteen octet header as part of each signalling message). The upcoming E-LOTOS standard provides libraries of data types, and this may provide a fruitful area of future research. In particular, these libraries may answer some of the concerns raised by the work of Fernández et al. [FQVM88], discussed on page 29 of this thesis.

Improve coupling in communication channels Section 3.8.3 discusses the coupling between actors and explains how a channel that loses messages also allows the recipient actor to ignore one or more messages. The current channel specifications do not allow the designer to define a channel that does not lose messages, but that allows the recipient to ignore messages that it cannot handle. Further research could explore the possibility of an alternative channel mechanism that would allow loose coupling without arbitrary message loss.

Explore use of SDL The thesis explores a design methodology integrating LOTOS and ROOM. Another interesting avenue of research would be to consider implementing specifications written using other formal techniques, such as SDL, in ROOM.

Extend validation method The validation sections of this thesis are intended to illustrate the possibility of validating LOTOS specifications written in the style described in section 3.6.1 against requirements. Applied to industrial problems, a much more rigorous process of validation and verification would be required.

Add traceability Future work could examine the traceability of designs created using this methodology. That is, it would be useful to designers to be able to trace elements of the requirements document through the LOTOS specification to the ROOM implementation.

Appendix A

Case Study Details

This appendix gives details of one of the case studies used to illustrate the design methodology; the alternating-bit protocol discussed in section 3.9. Section A.1 presents the whole LOTOS specification written for this thesis. This is followed in section A.2 by the ROOM model that was derived from this specification. Finally, section A.3 includes some of the C++ code automatically generated from the ROOM model.

A.1 LOTOS Specification of ABP

A LOTOS specification of the Alternating Bit Protocol was written following the style outlined in chapter 3. The important parts of the specification are as follows:

1. Lines 9 through 19 define the data type `message`. Objects of this type are composed of two elements: a number (the payload of the message) and an alternation bit. Lines 11 through 14 describe the possible operations on an object of type message, providing methods to construct a message from a number and a bit, and to extract the number and the bit from an existing message. Lines 14 through 18 provide equations to carry out these operations.

2. Lines 21 through 24 define the data type `altbit` which is a simple enumeration of two values.

3. Lines 26 through 39 define the data type `number` which is simply an enumeration of four values, 'zero' through 'three', together with some simple operations to find the predecessor and successor of a given number.

4. Lines 41 through 44 define the data type `timerSignal` which is a simple enumeration of values.

5. Lines 46 through 51 describe the behaviour of the specification, which consists of the process `abp_system` synchronising on gates `sendmsg` and `recvmsg` with the process `abp_tester`. Putting the tester in parallel with the system allows for execution of the system within the LMC model checker.

6. Lines 55 through 64 define the process `abp_system`, which is the top-level behaviour of the ABP system. This system is composed of a sender (`abp_s`) and a receiver (`abp_r`) communicating over a channel (`channel`). Note that the events occurring on the channel (`sendpdu`, `recvpdu`), `sendack` and `recvack`) are hidden using the `hide` operator. This hiding of events means that the only events visible from outside the `abp_system` are the `sendmsg` and `recvmsg` corresponding to the sending and reception of a message, respectively.

7. Lines 66 through 166 define the structure and behaviour of the sender, `abp_s`. The sender is decomposed into two elements (on lines 67 through 72): a state machine governing the behaviour of the sender with respect to sending messages and a timer.

   The sender's state machine (`abp_s_state_machine`) is defined on lines 74 through 138. The initial behaviour of `abp_s_state_machine` is simply to start in state `sready0`. This state corresponds to the sender being ready to send a message, and having its alternation bit set to zero. The behaviour of this state is defined on lines 77 through 85. Lines 78 through 81 indicate that if the sender receives a message to be sent on to the receiver, that it should create a message with alternation bit zero, send the message to the receiver and start the timer. The sender then moves to state `swaiting0` (line 81). Alternatively, if the sender receives an acknowledgement from the receiver (line 83), it can ignore it and remain in state `ready0`. Lines 87 through 95 define state `sready1`, which is exactly the same as `sready0` except for the value of the alternation bit.

   Lines 97 through 116 define the state `swaiting0`, which corresponds to the sender having sent a message to the receiver and waiting for an acknowledgement with alternation bit zero. Line 99 corresponds to the sender receiving an acknowledgement, and the following behaviour depends upon the value of the alternation bit. If the value is zero (ie. the expected value), the timer is stopped and the sender moves to state `sready1`, where it is ready to send another message. If the value is one, the receiver is assumed not to have received the message and the message is resent (line 106) and the timer restarted. The sender then remains in state `swaiting0`. Line 111 corresponds to the sender receiving a timeout message from the timer, which will happen if the receiver does not acknowledge a message within a given time period. The message is resent and the timer restarted. The sender then remains in state `swaiting0`, waiting for a correct acknowledgement. Lines 118 through 138 define state `swaiting1`, which is exactly the same as `swaiting0` except for the expected value of the alternation bit.

Lines 140 through 165 define the timer actor, which provides timing services to the sender's state machine. The timer starts in state `timerready`, indicating that it is ready to start timing. When a `timerStart` message is received, the timer moves to state `timertiming`. In this latter state, the timer may receive a message indicating that it should stop (line 158), which causes it to return to `timerready`. It may also receive an extraneous `timerStart` message, in which case it remains in the `timerready` state. Finally, it may generate a `timeout` message.

8. The receiver actor (lines 168 through 196) is much simpler than the sender, and does not contain a timer. The receiver starts in state `rwaiting0`, indicating that it is waiting for a message with alternation bit zero. State `rwaiting0` is defined on lines 171 through 182. If a message is received (line 172), an acknowledgement is sent with the same alternation bit as was in the message. The alternation bit is then checked to see whether its value is as expected. If so (line 175), the payload data of the message is passed on and the receiver moves to state `rwaiting1`. If the value is incorrect, the receiver remains in state `rwaiting0`, awaiting the arrival of a message with the correct alternation bit. State `rwaiting1` (lines 184 through 195) is the same as `rwaiting0` except for the expected value of the alternation bit.

9. The communication channel between sender and receiver is defined on lines 198 through 211. The channel consists of two identical unidirectional channels and the unidirectional channel is defined on lines 203 through 210. The channel may receive a message (line 204) and then either pass it on (line 206) or lose it (line 208).

10. The specification may be tested by one of three processes, appearing on lines 214 through 250. The first, `abp_tester` consists of a process (`abp_tester_s`) that injects messages into the sender interleaved with a process (`abp_tester_r`) that receives messages at the other end of the system. If both messages appear successfully, the enable on line 220 will be triggered and the special event `success` reached. Testers `abp_tester2` and `abp_tester3` are simpler processes that check that one or two messages can be sent over the ABP system. Although both `abp_tester` and `abp_tester2` check that two successive messages can be sent and received in order, the former interleaves the sending and receiving of messages so that both messages could be sent before the first is received, for

example. The second tester, abp_tester2 requires that the first message is received before the second is sent.

The specification was validated as described in section 3.9.3 and a ROOM model derived as described in section 3.9.4. This ROOM model is presented in the following section.

```
(*****************************************************)
(* An Alternate Bit Protocol Specification in LOTOS    *)
(* by Neil Hart                                        *)
(* Protocol Research Group, University of Ottawa      *)
(*****************************************************)

specification abp[sendmsg,recvmsg, success] :  noexit

  type message is number, altbit
  sorts mess
  opns data :  mess -> num
       seq :  mess -> bit
       msg :  num , bit -> mess
  eqns forall Data:  num, Seq:  bit
     ofsort num
       data(msg(Data,Seq))= Data;
     ofsort bit
       seq(msg(Data,Seq)) = Seq;
  endtype

  type altbit is
  sorts bit
  opns 0, 1 :-> bit
  endtype

  type number is
  sorts num
  opns zero, one, two, three :-> num
       succ :  num -> num
       pred :  num -> num
  eqns ofsort num
       succ(zero) = one;
       succ(one) = two;
       succ(two) = three;
```

```

          pred(three) = two;
          pred(two) = one;
          pred(one) = zero;
    endtype

    type timerSignals is
    sorts tsignal
    opns timerStart, timerStop, timeout :-> tsignal
    endtype

  behavior
      (
          abp_tester[sendmsg, recvmsg, success]
      |[sendmsg, recvmsg]|
          abp_system[sendmsg, recvmsg]
      )

  where

  process abp_system[sendmsg,recvmsg]:noexit:=
      hide sendpdu,recvpdu,sendack,recvack in
      (
          abp_s [sendmsg,sendpdu,recvack]
      |[sendpdu,recvack]|
          channel [sendpdu, recvpdu, sendack, recvack]
      |[sendack, recvpdu]|
          abp_r [recvmsg,sendack,recvpdu]
      )
  endproc

  process abp_s[smsg,spdu,rack]:noexit:=
      hide tsig in
      (
          abp_s_state_machine[smsg, spdu, rack, tsig]
      |[tsig]|
          timer[tsig]
      )
  where
```

```
process abp_s_state_machine[smsg,spdu,rack,tsig]:noexit:=
    sready0[smsg, spdu, rack, tsig]
where
    process sready0[smsg,spdu,rack,tsig]:noexit:=
        smsg ?n:num;
        spdu !msg(n, 0 of bit);
        tsig !timerStart;
        swaiting0[smsg, spdu, rack, tsig] (n)
    []
        rack ?X:mess;
        sready0[smsg, spdu, rack, tsig]
    endproc

    process sready1[smsg,spdu,rack,tsig]:noexit:=
        smsg ?n:num;
        spdu !msg(n, 1 of bit);
        tsig !timerStart;
        swaiting1[smsg, spdu, rack, tsig] (n)
    []
        rack ?X:mess;
        sready1[smsg, spdu, rack, tsig]
    endproc

    process swaiting0[smsg,spdu,rack,tsig](n:num):noexit:=
        (
            rack?X:mess;
            (
                [seq(X) = 0 of bit] ->
                  tsig !timerStop;
                  sready1[smsg, spdu, rack, tsig]
            []
                [seq(X) = 1 of bit] ->
                  spdu !msg(n, 0 of bit);
                  tsig !timerStart;
                  swaiting0[smsg, spdu, rack, tsig] (n)
            )
        []
            tsig !timeout;
            spdu !msg(n, 0 of bit);
```

```
                tsig !timerStart;
                swaitingO[smsg, spdu, rack, tsig] (n)
            )
        endproc

        process swaiting1[smsg,spdu,rack,tsig](n:num):noexit:=
            (
                rack?X:mess;
                (
                    [seq(X) = 1 of bit] ->
                      tsig !timerStop;
                      sready0[smsg, spdu, rack, tsig]
                []
                    [seq(X) = 0 of bit] ->
                      spdu !msg(n, 1 of bit);
                      tsig !timerStart;
                      swaiting1[smsg, spdu, rack, tsig] (n)
                )
            []
                tsig !timeout;
                spdu !msg(n, 1 of bit);
                tsig !timerStart;
                swaiting1[smsg, spdu, rack, tsig] (n)
            )
        endproc
    endproc

    process timer[tsig]:noexit:=
        timerready[tsig]
    where
        process timerready[tsig]:noexit:=
            (
                tsig !timerStart;
                timertiming[tsig]
            []
                tsig !timerStop;
                timerready[tsig]
            )
        endproc
```

```
                process timertiming[tsig]:noexit:=
                    (
                        tsig !timerStart;
                        timertiming[tsig]
                    []
                        tsig !timerStop;
                        timerready[tsig]
                    []
                        tsig !timeout;
                        timerready[tsig]
                    )
                endproc
            endproc
        endproc

        process abp_r[rmsg,sack,rpdu]:noexit:=
            rwaiting0[rmsg, sack, rpdu]
        where
            process rwaiting0[rmsg,sack,rpdu]:noexit:=
                rpdu?X:mess;
                sack!msg(zero,seq(X)) ;
                (
                    [seq(X) = 0 of bit] ->
                        rmsg!data(X);
                        rwaiting1[rmsg, sack, rpdu]
                []
                    [seq(X) = 1 of bit] ->
                        rwaiting0[rmsg, sack, rpdu]
                )
            endproc

            process rwaiting1[rmsg,sack,rpdu]:noexit:=
                rpdu?X:mess;
                sack!msg(zero,seq(X)) ;
                (
                    [seq(X) = 1 of bit] ->
                        rmsg!data(X);
                        rwaiting0[rmsg, sack, rpdu]
```

```
                    []
                [seq(X) = 0 of bit] ->
                    rwaiting1[rmsg, sack, rpdu]
            )
        endproc
    endproc

    process channel [in1,out1,in2,out2]:noexit:=
        chann [in1, out1]
        |||
        chann [in2, out2]
    where
        process chann [ing,outg]:noexit:=
            ing?X:mess ;
            (
                outg!X ; chann[ing,outg]
            []
                i; chann[ing, outg]
            )
        endproc
    endproc


    process abp_tester[sendmsg,recvmsg,success]:noexit:=
        (
            abp_tester_s [sendmsg]
            |||
            abp_tester_r [recvmsg]
        )
        >>
        success;
        stop
    where
        process abp_tester_s [s]:exit:=
            s !  one of num;
            s !  two of num;
            exit
        endproc
        process abp_tester_r [r]:exit:=
```

```
230            r !  one of num;
231            r !  two of num;
232            exit
233        endproc
234    endproc
235
236    process abp_tester2[sendmsg,recvmsg,success]:noexit:=
237        sendmsg !  one of num;
238        recvmsg !  one of num;
239        sendmsg !  two of num;
240        recvmsg !  two of num;
241        success;
242        stop
243    endproc
244
245    process abp_tester3[sendmsg,recvmsg,success]:noexit:=
246        sendmsg !  one of num;
247        recvmsg !  one of num;
248        success;
249        stop
250    endproc
251
252    endspec
```

A.2 ROOM Model of ABP

The derivation of the ROOM model from the LOTOS specification follows the process described in section 3.8. For example, the receiver is defined as:

```
168    process abp_r[rmsg,sack,rpdu]:noexit:=
```

The three gates listed (`rmsg`, `sack` and `rpdu`) require three ports on the corresponding ROOM actor. These ports may be seen in the structure diagram of `abp_r` in figure A.1.

As explained in section 3.8.3, the channel process in the LOTOS specification is not represented by an actor in the ROOM model, but instead by a binding between two actors using asynchronous message passing. Thus, the `abp_system` on lines 55 through 64 of the LOTOS specification is represented in ROOM as the structure in figure A.2.

The state machines representing the sender and receiver are transformed into ROOM as in figures A.3 and A.4.

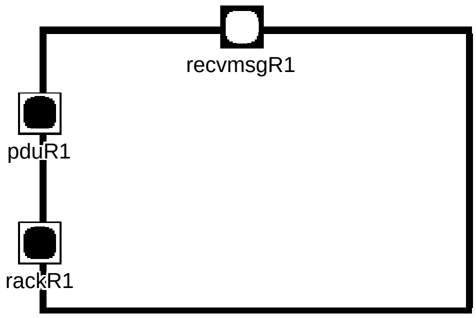


Figure A.1: ROOM representation of the abp_r actor.

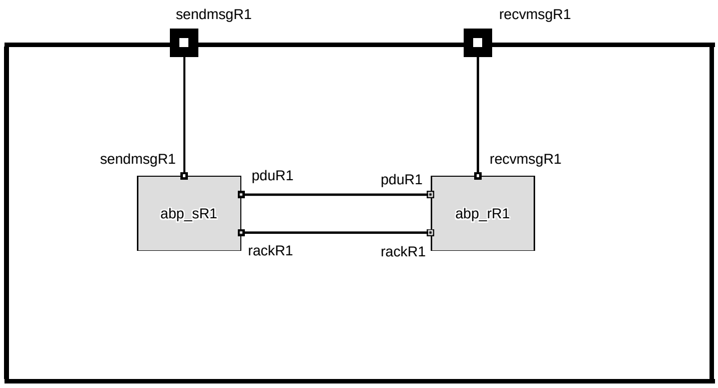


Figure A.2: ROOM representation of the ABP system.

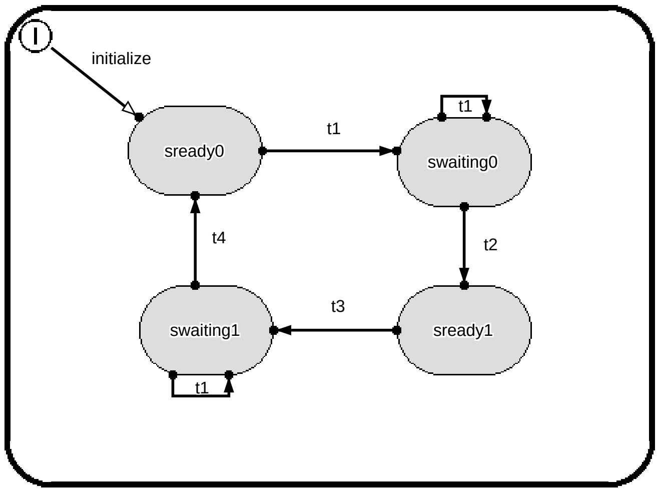


Figure A.3: Behaviour specification for the ABP sender actor in the ROOM notation.

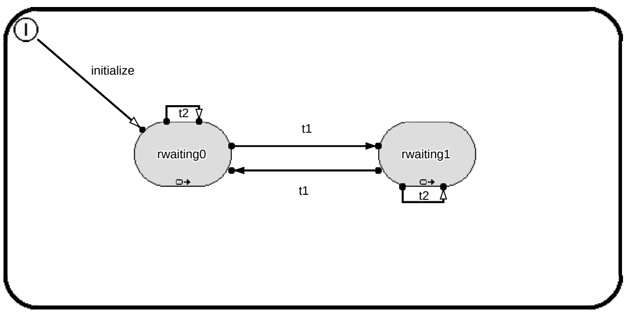


Figure A.4: Behaviour specification for the ABP receiver actor in the ROOM notation.

A.3 C++ Implementation of ABP

Much of the C++ implementation of the ROOM model is automatically generated by the ObjecTime toolset. The implementation includes a state machine derived from the graphical representation in the model. Apart from this automatically generated code, there are small segments of code written by the designer to implement, for example, guards on transitions or actions to be taken. In all, the C++ implementation of the ABP system was composed of around 1400 lines of C++, spread over about 40 source files. For brevity, only the source representing the `abp_s` actor will be presented here.

The first segments of code presented are those entered into code editors within the toolset to represent guard conditions on transitions. For example, the first segment returns a Boolean value indicating whether the alternation bit in the most-recently received message had the value `false`. This Boolean value is used by the automatically-generated state machine code to determine whether the corresponding transition should be taken.

```
********** start of file guard3_t2_event1.cc:

INLINE_METHODS int abp_s_Actor::guard3_t2_event1()
{
return ((message*)msg->data)->altbit == false;
}

********** end of file guard3_t2_event1.cc
********** start of file guard5_t4_event1.cc:

INLINE_METHODS int abp_s_Actor::guard5_t4_event1()
{
return ((message*)msg->data)->altbit == true;
}

********** end of file guard5_t4_event1.cc
********** start of file guard6_t1_event1.cc:

INLINE_METHODS int abp_s_Actor::guard6_t1_event1()
{
return ((message*)msg->data)->altbit == true;
}

********** end of file guard6_t1_event1.cc
********** start of file guard7_t1_event1.cc:
```

```
INLINE_METHODS int abp_s_Actor::guard7_t1_event1()
{
return ((message*)msg->data)->altbit == false;
}

********** end of file guard7_t1_event1.cc
```

The following segments of code provide the action code for transitions in the `abp_s` actor. For example, the first segment corresponds to the transition triggered by the arrival of data to be sent to the receiver. The data is stored (so that it can be retransmitted if necessary) and sent to the receiver through port `pduR1` with the alternation bit set to `false`.

```
********** start of file transition2_t1.cc:

INLINE_METHODS void abp_s_Actor::transition2_t1()
{
n = *RTDATA; // store incoming value
pduR1.send(spdu, message(n, false)); // send message
// start timer

;
}

********** end of file transition2_t1.cc
********** start of file transition3_t2.cc:

INLINE_METHODS void abp_s_Actor::transition3_t2()
{
// stop the timer
;
}

********** end of file transition3_t2.cc
********** start of file transition4_t3.cc:

INLINE_METHODS void abp_s_Actor::transition4_t3()
{
n = *RTDATA;
pduR1.send(spdu, message(n, true));
// start the timer
;
```

```
}

********** end of file transition4_t3.cc
********** start of file transition5_t4.cc:

INLINE_METHODS void abp_s_Actor::transition5_t4()
{
// stop the timer
;
}

********** end of file transition5_t4.cc
********** start of file transition6_t1.cc:

INLINE_METHODS void abp_s_Actor::transition6_t1()
{
pduR1.send(spdu, message(n, false));
}

********** end of file transition6_t1.cc
********** start of file transition7_t1.cc:

INLINE_METHODS void abp_s_Actor::transition7_t1()
{
pduR1.send(spdu, message(n, true));
}

********** end of file transition7_t1.cc
```

The remaining segments of code are all generated automatically and are used to implement the state machine and bind the user-entered code together.

```
********** start of file 1_abp_s_ActorBehavior.cc:

#ifdef PRAGMA
#pragma implementation "abp_s_Actor.h"
#endif
#include "abp_s_Actor.cc"
#undef Destructor
#undef SUPER
#define SUPER RTActor
#define Destructor abp_s_Actor_Destructor
```

```
#define RTDATA ((const RTDataObject*)msg->data)
#include "initBehavior.cc"

#undef RTDATA

#define CALLSUPER SUPER::guard3_t2_event1()
#define RTDATA ((const message*)msg->data)
#include "guard3_t2_event1.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::guard5_t4_event1()
#define RTDATA ((const message*)msg->data)
#include "guard5_t4_event1.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::guard6_t1_event1()
#define RTDATA ((const message*)msg->data)
#include "guard6_t1_event1.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::guard7_t1_event1()
#define RTDATA ((const message*)msg->data)
#include "guard7_t1_event1.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::transition2_t1()
#define RTDATA ((const num*)msg->data)
#include "transition2_t1.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::transition3_t2()
#define RTDATA ((const message*)msg->data)
#include "transition3_t2.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::transition4_t3()
#define RTDATA ((const num*)msg->data)
#include "transition4_t3.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::transition5_t4()
```

```
#define RTDATA ((const message*)msg->data)
#include "transition5_t4.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::transition6_t1()
#define RTDATA ((const message*)msg->data)
#include "transition6_t1.cc"
#undef RTDATA
#undef CALLSUPER
#define CALLSUPER SUPER::transition7_t1()
#define RTDATA ((const message*)msg->data)
#include "transition7_t1.cc"
#undef RTDATA
#undef CALLSUPER
#undef Destructor
#undef SUPER

********** end of file 1_abp_s_ActorBehavior.cc
********** start of file abp_s_Actor.cc:

#include "abp_s_Actor.h"
#include <initData.h>

#undef Destructor
#undef SUPER
#define SUPER RTActor
#define Destructor abp_s_Actor_Destructor

const RTObject_class abp_s_Actor::classData(&RTActor::classData,"abp_s");

const RTObject_class*abp_s_Actor::getClassData()const
{
return&abp_s_Actor::classData;
}

const RTStateId abp_s_Actor::_parent_state[]=
{
0,
0,
1,
1,
1,
```

```
1
};

const RTActor_class*abp_s_Actor::getActorData()const
{
static const char*const _state_names[]=
{
(char*)0,
"top",
"sready0",
"swaiting0",
"swaiting1",
"sready1"
};
static const RTFieldOffset _port_offsets[]=
{
RTOffsetOf(abp_s_Actor,sendmsgR1),
RTOffsetOf(abp_s_Actor,pduR1),
RTOffsetOf(abp_s_Actor,rackR1),
};
static const char*const _port_names[]=
{
"sendmsgR1",
"pduR1",
"rackR1",
};
static const RTFieldOffset _relay_offsets[]=
{
RTOffsetOf(abp_s_Actor,sendmsgR1_relay),
RTOffsetOf(abp_s_Actor,pduR1_relay),
RTOffsetOf(abp_s_Actor,rackR1_relay),
};
static const char*const _relay_names[]=
{
"sendmsgR1",
"pduR1",
"rackR1",
};
static const RTFieldOffset _ESV_offsets[]=
{
RTOffsetOf(abp_s_Actor,n),
};
```

```
static const void*_ESV_types[]=
{
(void*)1,
};
static const char*const _ESV_names[]=
{
"n",
};
static RTActor_class _info=
{
5,_state_names,abp_s_Actor::_parent_state,
//components
0,(RTComponentDescriptor*)0,
//ports
{3,_port_offsets,_port_names,(RTFieldType*)0},
//relays
{3,_relay_offsets,_relay_names,(RTFieldType*)0},
//ESVs
{1,_ESV_offsets,_ESV_names,(const RTFieldType*)_ESV_types},
//SAPs
{0,(RTFieldOffset*)0,(char const*const*)0,(RTFieldType*)0},
//SPPs
{0,(RTFieldOffset*)0,(char const*const*)0,(RTFieldType*)0},
(FsmStaticData*)0
};
if(_info.FsmData==(FsmStaticData*)0)_info.FsmData=initBehavior();
return&_info;
}

RTActor*new_abp_s_Actor(RTController*_rts,RTActorRef&_ref)
{return new abp_s_Actor(_rts,_ref);}

abp_s_Actor::abp_s_Actor(RTController*_rts,RTActorRef&_ref)
:RTActor(_rts,_ref)
,pduR1(this,"pduR1",2,1)
,rackR1(this,"rackR1",3,1)
,sendmsgR1(this,"sendmsgR1",1,1)
{
(num*)0; // for type error detection only
}

abp_s_Actor::~abp_s_Actor(void)
```

```
{
}

#undef Destructor
#undef SUPER

********** end of file abp_s_Actor.cc
********** start of file abp_s_Actor.h:

#ifndef __abp_s_Actor_h__
#define __abp_s_Actor_h__ included

#include "../system/RTSystem.h"
#ifdef PRAGMA
#pragma interface
#endif
#include "initData_simple.h"
#undef Destructor
#undef SUPER
#define SUPER RTActor
#define Destructor abp_s_Actor_Destructor

extern RTActor*new_abp_s_Actor(RTController*,RTActorRef&);

class abp_s_Actor:public RTActor
{
public:
RTAsyncCommSAP pduR1, rackR1, sendmsgR1;
RTRelaySAP pduR1_relay, rackR1_relay, sendmsgR1_relay;
abp_s_Actor(RTController*,RTActorRef&);
virtual~abp_s_Actor(void);
static const RTObject_class classData;
virtual const RTObject_class*getClassData(void)const;
virtual const RTActor_class*getActorData(void)const;
static const RTStateId _parent_state[];
static const FsmStaticData*initBehavior(void);
#line 500
INLINE_METHODS int guard3_t2_event1(void);
INLINE_METHODS int guard5_t4_event1(void);
INLINE_METHODS int guard6_t1_event1(void);
INLINE_METHODS int guard7_t1_event1(void);
INLINE_METHODS void transition2_t1(void);
```

```
INLINE_METHODS void transition3_t2(void);
INLINE_METHODS void transition4_t3(void);
INLINE_METHODS void transition5_t4(void);
INLINE_METHODS void transition6_t1(void);
INLINE_METHODS void transition7_t1(void);
#line 1000
protected:
num n;
#line 1500
#line 2000
private:
#line 2500
};

#undef Destructor
#undef SUPER
#endif

********** end of file abp_s_Actor.h
********** start of file initBehavior.cc:

const FsmStaticData*abp_s_Actor::initBehavior(void)
{
//  create a FsmStaticData instance and add methods from the superclass
FsmStaticData*_info=new FsmStaticData("abp_s_Actor",
                                        _CONVERT_ACTORIDS_(1),"RTActor");

//  addSAP(SAP name, SAP id, replication factor)

//  addMethod(method name, method pointer)
_info->addBMethod("guard3_t2_event1",
                         (RTActorBooleanFunc)abp_s_Actor::guard3_t2_event1);
_info->addBMethod("guard5_t4_event1",
                         (RTActorBooleanFunc)abp_s_Actor::guard5_t4_event1);
_info->addBMethod("guard6_t1_event1",
                         (RTActorBooleanFunc)abp_s_Actor::guard6_t1_event1);
_info->addBMethod("guard7_t1_event1",
                         (RTActorBooleanFunc)abp_s_Actor::guard7_t1_event1);
_info->addMethod("transition2_t1",
                         (RTActorDataObjectFunc)abp_s_Actor::transition2_t1);
_info->addMethod("transition3_t2",
                         (RTActorDataObjectFunc)abp_s_Actor::transition3_t2);
```

```
_info->addMethod("transition4_t3",
                         (RTActorDataObjectFunc)abp_s_Actor::transition4_t3);
_info->addMethod("transition5_t4",
                         (RTActorDataObjectFunc)abp_s_Actor::transition5_t4);
_info->addMethod("transition6_t1",
                         (RTActorDataObjectFunc)abp_s_Actor::transition6_t1);
_info->addMethod("transition7_t1",
                         (RTActorDataObjectFunc)abp_s_Actor::transition7_t1);

return _info;
}

********** end of file initBehavior.cc
```

Bibliography

[Amy93] Daniel Amyot. From LOTOS Specification to Occam Implementation. Submitted as dissertation for University of Ottawa course CSI7170, February 1993.

[Ash92] Pierre Ashkar. Symbolic Execution of LOTOS Specifications. Master's thesis, Department of Computer Science, University of Ottawa, 1992.

[AVACV93] A. Azcorra, E. Vázquez, M. Alvarez-Campana, and J. Vinyes. Formal Description Techniques at Work: An ISDN Q.931 Implementation Using LOTOS. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Specification, Testing and Verification, XIII*, 175–189. Elsevier Science Publishers B.V., 1993.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BG92] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall, Second edition, 1992.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, September 1988.

[Bol92] Tommaso Bolognesi. Catalogue of LOTOS Correctness Preserving Transformations, April 1992. ESPRIT II Lotosphere Project, Lo/WP1/T1.2/N0045/V03, Final Deliverable.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Second edition, 1994.

[Bri89] Ed Brinksma. A theory for the derivation of tests. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, 235–247. Elsevier Science Publishers B.V., 1989.

[BSS87] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. Lotos Specifications, their Implementations and their Tests. In B. Sarikaya and G.V. Bochmann, editors, *Protocol Specification, Testing and Validation, VI*, 349–360. Elsevier Science Publishers B.V., 1987.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 12(5):260–261, May 1969.

[BvdLV95] Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.

[BW97] Götz Brasche and Bernhard Walke. Concepts, Services and Protocols of the New GSM Phase 2+ General Packet Radio Service. *IEEE Communications*, 94–104, August 1997.

[CDH+96] J.P. Courtiat, P. Dembinski, G. Holzmann, L. Logrippo, R. Rudin, and P. Zave. Formal Methods after 15 years: Status and Trends. *Computer Networks and ISDN Systems*, 28:1845–1855, 1996.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CG97] Jian Cai and David J. Goodman. General Packet Radio Service in GSM. *IEEE Communications*, 122–131, October 1997.

[CK94] R. Castanet and O. Koné. Deriving Coordinated Testers for Interoperability. In O. Rafiq, editor, *Protocol Test Systems, VI*, 331–345. Elsevier Science Publishers B.V., 1994.

[CL91] Elspeth Cusack and Michael Lai. Object-Oriented Specification in LOTOS and Z, or My Cat Really is Object-Oriented. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, 179–202. Springer-Verlag, 1991.

[CM97a] R.G. Clark and A.M.D. Moreira. Formal user-centered models. In Howard Bowman and John Derrick, editors, *Formal Methods for Open Object-based Distributed Systems*, volume 2, 215–230. Chapman & Hall, 1997.

[CM97b] Robert G. Clark and Ana M. D. Moreira. Using a Formal User-Centred Model to build a Formal System-Centred Model. Technical Report CSM-140, Department of Computing Science and Mathematics, University of Stirling, Scotland, March 1997.

[Cri94] M. Crispin. Internet Message Access Protocol — Version 4. Available by anonymous ftp from `ftp://nic.ddn.mil/rfc/rfc1730.txt`, December 1994. RFC 1730.

[CRS90] Elspeth Cusack, Steve Rudkin, and Chris Smith. An object oriented interpretation of LOTOS. In S.T. Vuong, editor, *Formal Description Techniques, II*, 211–226. Elsevier Science Publishers B.V., 1990.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-state Verification. In *2nd Workshop on Formal Methods in Software Practice*, March 1998.

[dMAQM95] Tomas de Miguel, Arturo Azcorra, Juan Quemada, and José A. Mañas. A pragmatic approach to verification, validation and compilation. In Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors, *LOTOSphere: Software Development with LOTOS*, chapter 12, 235–253. Kluwer Academic Publishers, 1995.

[Dub90] Eric Dubuis. An Algorithm for Translating LOTOS Behavior Expressions into Automata and Ports. In S.T. Vuong, editor, *Formal Description Techniques, II*, 163–177. Elsevier Science Publishers B.V., 1990.

[EHM93] Patrik Ernberg, Thomas Hovander, and Francisco Monfort. Specification and Implementation of an ISDN Telephone System Using LOTOS. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, 171–186. Elsevier Science Publishers B.V., 1993.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag, 1985.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Methods and Semantics*, chapter 16, 995–1072. Elsevier Science Publishers B.V., 1990.

[FLS90] M. Faci, L. Logrippo, and B. Stépien. Formal Specification of Telephone Systems in LOTOS. In E. Brinksma, G. Scollo, and C.A. Vissers, editors, *Protocol Specification, Testing and Validation, IX*, 25–34. Elsevier Science Publishers B.V., 1990.

[FMVQ92] A. Fernández, C. Miguel, L. Vidaller, and J. Quemada. Development of Satellite Communication Networks based on LOTOS. In R.J. Linn, Jr. and M.Ü. Uyar, editors, *Protocol Specification, Testing and Validation, XII*, 179–192. Elsevier Science Publishers B.V., 1992.

[FQVM88] A. Fernández, J. Quemada, L. Vidaller, and C. Miguel. PRODAT – The Derivation of an Implementation from its LOTOS Formal Specification. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Validation, VIII*, 411–419. Elsevier Science Publishers B.V., 1988.

[fS89] International Organization for Standardization. Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, ISO/IEC 8807, 1989.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Ghr92] Brahim Ghribi. A model checker for LOTOS. Master's thesis, Department of Computer Science, University of Ottawa, 1992.

[GL93] B. Ghribi and L. Logrippo. A Validation Environment for LOTOS. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Specification, Testing and Verification, XIII*, 93–108. Elsevier Science Publishers B.V., 1993.

[gsm97] Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Service description; Stage 2 (GSM 03.60 version 5.1.0). Draft standards document of the European Telecommunications Standards Institute, October 1997.

[gtp97] Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); GPRS Tunnelling Protocol (GTP) across the Gn and Gp Interface; (GSM 09.60 proposed version 1.2.0). Draft standards document of the European Telecommunications Standards Institute, September 1997.

[Hal90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 11–19, September 1990.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hed93] Mikael Hedlund. The Integration of LOTOS with an Object Oriented Development Method. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, 73–82. Springer-Verlag, 1993.

[HF97] Christian Hernalsteen and Arnaud Février. Introduction of a Suspend/Resume Operator in ET-LOTOS. In Miguel Bertran and Teodor Rus, editors, *Transformation-Based Reactive Systems Development*, number 1231 in Lecture Notes in Computer Science, 400–414. Springer, May 1997.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[JK90] Bengt Jonsson and Ahmed Hussain Khan. Implementing a Model Checking Algorithm by Adapting Existing Automated Tools. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, 179–188. Springer-Verlag, 1990.

[KBG93] Günter Karjoth, Carl Binding, and Jan Gustafsson. LOEWE: A LOTOS engineering workbench. *Computer Networks and ISDN Systems*, 25(7):853–874, February 1993.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Second edition, 1988.

[Kre95] H. Kremer. Derivation of efficient implementations from formal descriptions – issues, methods and conformance. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII*, 431–446. Chapman & Hall, 1995.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[LFHH92] L. Logrippo, M. Faci, and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23(5):325–342, February 1992.

[LYS+93] Gonzalo León, Juan C. Yelmo, Carlos Sánchez, F. Javier Carrasco, and Juan J. Gill. An Industrial Experience on LOTOS-Based Prototyping for Switching Systems Design. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, 83–92. Springer-Verlag, 1993.

[May89] Thomas Mayr. Specification of Object-Oriented Systems in LOTOS. In K.J. Turner, editor, *Formal Description Techniques*, 107–119. Elsevier Science Publishers B.V., 1989.

[MdM89] J.A. Mañas and T. de Miguel. From LOTOS to C. In K.J. Turner, editor, *Formal Description Techniques*, 79–84. Elsevier Science Publishers B.V., 1989.

[MdMSA93] José A. Mañas, Tomás de Miguel, Joaquín Salvachúa, and Arturo Azcorra. Tool support to implement LOTOS formal specifications. *Computer Networks and ISDN Systems*, 25(7):815–839, February 1993.

[MdMvT89] José Mañas, Tomás de Miguel, and Huub van Thienen. The Implementation of a Specification Language for OSI Systems. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, 409–421. Elsevier Science Publishers B.V., 1989.

[Meh96] Asha Mehrotra. *GSM System Engineering.* Artech House, Inc., 1996.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MR96] J. Myers and M. Rose. Post Office Protocol - Version 3. Available by anonymous ftp from `ftp://nic.ddn.mil/rfc/rfc1939.txt`, May 1996. STD 53, RFC 1939.

[Nis97] Nimal Nissanke. *Realtime Systems.* Prentice-Hall, 1997.

[Pos80] J. Postel. User Datagram Protocol. Available by anonymous ftp from `ftp://nic.ddn.mil/rfc/rfc768.txt`, August 1980. RFC 768, ISI.

[Pos81a] J. Postel. Internet Protocol — DARPA Internet Program Protocol Specification. Available from anonymous ftp from `ftp://nic.ddn.mil/rfc/rfc791.txt`, September 1981. RFC 791, USC/Information Sciences Institute.

[Pos81b] Jon Postel. Transmission Control Protocol — DARPA Internet Program Protocol Specification. Available by anonymous ftp from `ftp://nic.ddn.mil/rfc/rfc793.txt`, September 1981. RFC 793, USC/Information Sciences Institute.

[Pre87] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, Second edition, 1987.

[PYD95] A. Petrenko, N. Yevtushenko, and R. Dssouli. Testing Strategies for Communicating FSMs. In Tadanori Mizuno, Teruo Higashino, and Norio Shiratori, editors, *Protocol Test Systems, VII*, 193–208. Chapman & Hall, 1995.

[QAP95] Juan Quemada, Arturo Azcorra, and Santiago Pavón. The LOTOSphere Design Methodology. In Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors, *LOTOSphere: Software Development with LOTOS*, chapter 2, 29–58. Kluwer Academic Publishers, 1995.

[QPF89] Juan Quemada, Santiago Pavón, and Angel Fernández. Transforming LOTOS Specifications with Lola: The Parameterized Expansion. In K.J. Turner, editor, *Formal Description Techniques*, 45–54. Elsevier Science Publishers B.V., 1989.

[QPF90] Juan Quemada, Santiago Pavón, and Angel Fernández. State Exploration by Transformation with LOLA. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, 294–302. Springer-Verlag, 1990.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Riv92] R. Rivest. The MD5 Message-Digest Algorithm. Available by anonymous ftp from `ftp://nic.ddn.mil/rfc/rfc1321.txt`, April 1992. RFC 1321.

[Rud92] Steve Rudkin. Inheritance in LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques, IV*, 409–424. Elsevier Science Publishers B.V., 1992.

[Sco95] John Scourias. Overview of the Global System for Mobile Communications. `http://ccnga.waterloo.ca/~jscouria/GSM/gsmreport.ps`, May 1995.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[SL93] Bernard Stepien and Luigi Logrippo. Status-Oriented Telephone Service Specification: An Exercise in LOTOS Style. Technical Report TR-93-07, Telecommunications Software Engineering Research Group, Department of Computer Science, University of Ottawa, February 1993.

[SL94] B. Stepien and L. Logrippo. Status-Oriented Telephone Service Specification. In T.Rus and C.Rattray, editors, *Theories and Experiences for Real-Time System Development*, volume 2 of *AMAST Series in Computing*, 265–286. World Scientific, 1994.

[SP95] Jeroen Schot and Luís Ferreira Pires. Design and Implementation Strategies. In Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors, *LOTOSphere: Software Development with LOTOS*, chapter 3, 59–85. Kluwer Academic Publishers, 1995.

[Ste94] Bernard Stepien. Specifying Suspend-Resume Behaviour in LOTOS. *LOTOS News*, 2:20–24, August 1994.

[Tre89] J. Tretmans. HIPPO: A LOTOS Simulator. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, 391–396. Elsevier Science Publishers B.V., 1989.

[Tur93] Kenneth J. Turner. *Using Formal Description Techniques: An Introduction to ESTELLE, LOTOS and SDL*. John Wiley & Sons, 1993.

[vEKvS90] Peter van Eijk, Harro Kremer, and Marten van Sinderen. On the use of specification styles for automated protocol implementation from LOTOS to C. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Validation, X*, 157–168. Elsevier Science Publishers B.V., 1990.

[VPvdL95] Chris A. Vissers, Luís Ferreira Pires, and Jeroen van de Lagemaat. Lotosphere, an attempt towards a Design Culture. In Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors, *LOTOSphere: Software Development with LOTOS*, chapter 1, 3–28. Kluwer Academic Publishers, 1995.

[VSvSB91] Chris A. Vissers, Giuseppe Scollo, Marten van Sinderen, and Ed Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.

[VvSP93] Chris A. Vissers, Marten van Sinderen, and Luís Ferreira Pires. What Makes Industries Believe in Formal Methods. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Specification, Testing and Verification, XIII*, 3–26. Elsevier Science Publishers B.V., 1993.

[WD95] Ken Warkentyne and Eric Dubuis. The COLOS Compiler. In Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors, *LOTOSphere: Software Development with LOTOS*, chapter 16, 319–331. Kluwer Academic Publishers, 1995.

[Weg86] Peter Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, October 1986.

[Wie95] Edwin Wiedmer. Lotos Industrial Applications. In Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors, *LOTOSphere: Software Development with LOTOS*, chapter 5, 109–120. Kluwer Academic Publishers, 1995.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[YHA+96] Keiichi Yasumoto, Teruo Higashino, Kota Abe, Toshio Matsuura, and Kenichi Taniguchi. A LOTOS Compiler Generating Multi-threaded Object Codes. In Gregor von Bochmann, Rachida Dssouli, and Omar Rafiq, editors, *Formal Description Techniques, VIII*, 271–286. Chapman & Hall, 1996.

[ZWR+80] Pitro Zafiropulo, Colin H. West, Harry Rudin, D.D. Cowan, and Daniel Brand. Towards Analyzing and Synthesizing Protocols. *IEEE Transactions on Communications*, COM-28(4):651–661, April 1980.