EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science

MASTER'S THESIS

The Specification and Validation of the OM/RR-Protocol

by

T.A.C. Willemse

Supervisor : Prof.dr. J.C.M. Baeten
Advisors : Ir. A. Klomp and Dr.ir J.Tretmans

June 1998

Abstract

Formal Methods offer the means for validating and verifying complex systems. Furthermore, concise and unambiguous specifications can be written using Formal Description Techniques.
This thesis deals with the formal specification of a data communications protocol. This protocol is to be used in a system called the *Operator Support System* (OSS). The specifications and the models for these specification are described in this thesis. Several unclarities and omissions were found in the documents describing this protocol. Furthermore, subsequent validation and analysis of these specifications and models, using two toolboxes, viz. LITE and EUCALYPTUS is discussed.

Acknowledgements

I would like to thank everybody at CMG for supporting me during my traineeship, especially everybody at the SIG-MCS/FM group for sharing their knowledge and experiences with me. Furthermore, I would like to thank all my friends and relatives for their support during the easier and more difficult periods of my traineeship. I especially appreciate the interest Laurens put into my work and the effort he made in reading and commenting on several parts of this thesis.
Special thanks go to Jan Tretmans from the University of Twente and Arjen Klomp from CMG, who both assisted me and had to put up with the huge amount of paperwork I produced during this traineeship. They both had a major influence on this thesis by providing me with guidelines and numerous suggestions for improvement of my work.
Finally, I would like to thank Jos Baeten of the Eindhoven University of Technology for assisting me during this traineeship and without whom this traineeship would never have been possible.

The Hague, June 1998
Tim Willemse

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is the result of a nine months traineeship at CMG Advanced Technology Government in The Hague. It deals with the specification and validation of a data communications protocol, used in the *Operator Support System* (OSS), a system described by the *Adviesdienst voor Verkeer en Vervoer* (AVV). This Chapter is divided into several Sections: Section 1.1 discusses the system OSS, Section 1.2 discusses issues related to protocols such as the concept of service for a protocol. Section 1.3 highlights several aspects concerning the choice for a suitable Formal Description Technique (FDT), Section 1.4 addresses different styles for writing formal specifications and Section 1.5 finally, discusses the objective for the formal specification. Reading directions for the rest of this thesis can be found in Section 1.6.

1.1 The Operator Support System

The Operator Support System (OSS) is a distributed system to support traffic operators and traffic managers at operating centres. In the past years, various Motorway Management Systems (MMSs) have been developed, such as fog detection, congestion detection, video systems for observation and variable message signs for controlling trafic. For historical reasons, every MMS had its own unique specification and implementation, thus gradually leading to distributed systems, each with their own interfaces and functionalities.

The actual controlling of these systems takes place at an Operating Centre (OC). There are several OCs throughout the country, and every OC has a clear, own domain which it controls. Nowadays, however, more and more cooperation is demanded between these OCs, and the strict division in domains is limiting this cooperation.

In recent years this awareness has grown, and this led to the description of a system called OSS. This system is supposed to be used in the near future to integrate the independent OCs and thereby to increase the power of the system regarded as a whole. The idea is, to enable one OC to take over duties from another OC, or to perform certain tasks that at some point exceeds the OC's own domain. One can think of, for instance, an operator from centre A reporting a traffic jam, and in order to find out how many kilometres this congestion is, he needs information only centre B can register. It would be useful to by-pass centre B, and access the information directly. Another aspect that will be possible, is to have one centre completely being taken over by another centre, for example at nights, or when there are very few controlling tasks to perform.

Although all this may sound fairly simple, reality learns that when a complex distributed system, such as the OSS, is implemented, many problems arise from the sheer fact that even to those who devised the system not every problem is foreseen.
In such cases, it helps to apply some strict design principles, such as *orthogonality* and *separation*

of concerns, that help divide the problem into managable pieces. Two important parts of the OSS were determined to be:

- the *data communications protocols.*
- the *authorization protocol.*

Authorization protocol. Basically the authorization protocol was devised to determine *when* data communications between different objects could take place, and as such can be regarded as a protocol utilizing the service of the data communications protocol.

Data communications protocol. The second, and perhaps even more important protocol, is the data communications protocol, since this has to function as the backbone of the OSS: not only does every OC have to communicate with its MMSs, but the communications between two OCs has to be captured as well using the same data communications protocol. This data communications protocol was christened the *OM/RR protocol.*

The OM/RR-protocol is, according to [AVV97], a two layered data communications protocol, loosely modelled after the OSI's CMIS [ISO90b], CMIP [ISO90a] and some parts of System Management [ISO90d] for the OM-layer, and OSI Remote Operation [ISO90c] for the RR-layer.
A problem that was encountered, reading the documents concerning the OSS project, was the brief documentation of the OM/RR-protocol. Especially the lack of specifications describing the dynamic behaviour of the OM/RR-protocol, such as a service specification, would make it hard to get a clear picture of the reliability and functionality of the OSS as a whole.

In essence data communications protocols are very hard to specify unambiguously using natural language. However, they do lend themselves excellently for a formal description using a Formal Description Technique, and as such, this thesis set out to do so.

1.2 Protocols and Services

Although not always acknowledged, the service concept is one of the major issues when specifying a (communication) protocol. Not only is it of concern to the service users, it also is necessary to build correctness proofs and discussing the correctness of the system's design.

In order to discuss the implications and the advantages of the service concept, it first has to be defined. However, no formal model of the concept of service shall be given, since this is beyond the scope of this thesis.

The model used in this thesis for protocol specification relies on two classes of protocol entities: the *service users* and the *service provider.* The communication between protocol entities follows strict rules, commonly refered to as the *protocol.* Figure 1.1 depicts the relation between these classes.

If two protocol entities are to communicate via an underlying service provider, a protocol entity utilizes *service primitives.* A service primitive can be considered as an elementary interaction between a service user and the service provider during which certain values for the various parameters of the primitive are established to which both user and provider can refer. Note that the execution of a service primitive does not imply passing of information in only one direction, but allows for a much richer variety of parameter value establishment than value passing alone: e.g. value checking and value negotiation are possibilities as well. The *Service Access Point* (SAP) is the common boundary of a service user and the service provider and is where the interactions of service primitives are executed.

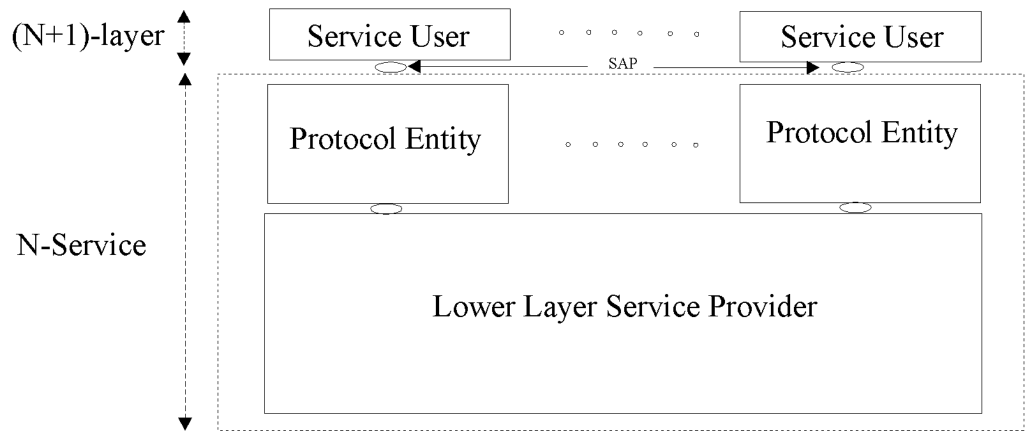

Figure 1.1: *The service model used in this report*

One can regard the service provider as an abstract machine, accessible from a number of SAPs. The execution of a service primitive at one SAP usually invokes the execution of another service primitive at (another) SAP whose parameter values may depend on the parameter values of the invoking primitive. Furthermore, the service provider is capable of spontaneous *internal actions*, leading to the execution of one or more service primitives at SAPs. Thus one way of specifying a service is expressed in terms of the possible orderings of service primitives observed at the SAPs and their parameter value dependencies. This kind of specification is often refered to as an *observational*, or *extensional* specification, as it does not reflect the internal structure of the service provider, yet only defines the behaviour of the provider as it can be observed by the users.

Another approach to defining the service is the *generative*, or *intensional* approach, in which the internal structure of the service provider is revealed. Instead of regarding the service provider as a black box, the service provider is decomposed one layer into (N)-protocol entities, which communicate using another *lower-layer* (N-1)-service provider. This approach is most often encountered in multi-layered protocol architectures. In this report, this decomposition is refered to as the (N)-protocol specification. Henceforth, when discussing the (N)-service, the extensional approach is meant, whereas when discussing the (N)-protocol, the intensional approach is meant.

As is to be expected, the intensional approach is often much more elaborate than the extensional approach.
Much of the information presented in this Section is taken from [VL86].

1.3 Formal Description Techniques

When describing a protocol formally, the first question is which formalism is to be used. The answer to this question depends mainly on what the goals are for the formal description, and the nature of the protocol.
The criteria of the FDT that shall be used for the formal description of the OM/RR-protocol are as follows:

1. The FDT must allow for a concise and unambiguous specification of both the OM-service and the OM-protocol (i.e. it must be able to model dynamic behaviour).

2. The FDT must be supported by tools that simulate and test and if possible, proof properties of both the OM-service and the OM-protocol.

The number of FDTs that adhere to these criteria is limited: not every FDT is able to express the dynamic behaviour (e.g. Z), or has an unambiguous (accepted) semantics associated to it (e.g. SDL). Furthermore, not every FDT is supported by tools (e.g. ACP). The FDT PROMELA is mainly used for modelling instead of specifying, has no (accepted) semantics associated to it and as such was regarded unsuitable.
Another restriction that played a part in the choice for a suitable FDT, was the availability of supporting tools at CMG, combined with the knowledge of how to deal with these tools and the FDT. This narrowed the number of suitable FDTs down to only one: the FDT LOTOS [BB88, ISO88].

1.4 Specification Styles

There exist several styles for writing formal specifications for systems. Each style has its own advantages and disadvantages, and as such, it is vital to choose the right style or combination of styles at the right time. The four major specification styles are *monolythical, state-oriented, constraint-oriented* and *resource-oriented*. A brief characterization is given below.

Monolythical style. The monolythical style can be characterized as a style in which only observable interactions are presented and ordered as a collection of alternative sequences of interactions in branching time. This style, however, is only suitable up to a limited complexity of the system, and is therefore mostly used for very simple systems.

Constraint-oriented style. In the constraint-oriented style only observable interactions are presented. Their temporal ordering is defined by a conjunction of different constraints. This style is very suitable for extensional descriptions.

State-oriented style. In this style, the system that is to be described, is regarded as a single resource whose internal state space is explicitly defined. This style clearly shows the amount of state information to be maintained by a resource and the complexity of the manipulation of this information.

Resource-oriented style. In the resource-oriented style both the observable and the internal interactions are presented. The behaviour in terms of the observable interactions is defined by a composition of separate resources in which the internal interactions are hidden. These resources may again be specified using any style.

For a more extensive discussion on the differences and the situations when to use which style, the reader is refered to [VSvSB91]. The styles mainly used in the LOTOS-specifications, discussed in this thesis are the resource-oriented style and the constraint-oriented style.

1.5 Analysis and Validation of the OM/RR-protocol

Once a formal specification of the OM-service and the OM-protocol (i.e. The OM-protocol entities and the RR-service) is present, several aspects have to be analysed: the major issues are whether both specifications are free of deadlock and livelock, and whether there exists an equivalence or a preorder relation between the observable behaviour of the OM-protocol on the one hand, and the OM-service on the other hand. Since the specifications are too large to prove equivalent, or analyse by hand, two toolboxes have been used, viz. LITE and EUCALYPTUS [Gar96]. This, however, means that several simplifications with respect to the formal specifications will have to be introduced. The OM-service specification and the OM-protocol, found in Appendices A and B, are modelled to accomodate for the restrictions the toolboxes pose on LOTOS-descriptions. These models are described in detail in Chapters 2, 3 and 4, and are subject of analysis and validation.

1.6 Thesis Outline

This thesis is divided into six Chapters and three Appendices. The OM-service model is described in Chapter 2, which discusses the service elements used in the service and contains the dynamic aspects of the OM-service described in LOTOS. Chapter 3 in turn discusses the service elements used in the lower layer service — the RR-service layer — and contains the dynamic aspects of the RR-service described in LOTOS. The OM-protocol entity model is presented in Chapter 4. The use of tools and validation aspects are discussed in Chapter 5 Finally, Chapter 6 discusses several suggestive remarks and impressions gained in the traineeship. Appendix A contains the OM-service specification and Appendix B contains the OM-protocol specification.

Chapter 2

The OM-service Model

In this chapter, the OM-service model is presented. First, the environment in which the OM-service is to function is discussed in Section 2.1, then the service elements and service primitives are determined in Section 2.2. Finally, in Section 2.3, the dynamic behaviour of the OM-service model will be described using LOTOS.

2.1 The OM-service Provider and the OM-service Users

The service users for the OM/RR-service are, according to [AVV97], *Managed Objects.* The definition for a Managed Object is:

Definition 2.1 (Managed Object) *A Managed Object is the management view of a resource.*

This definition, however, will not have any impact on the work presented here, as it is too obscure and in essence can mean anything.

In essence, the OM/RR-protocol is a two-layered communication protocol. Since the OM-protocol entity is designed to utilize the service offered by the RR-service layer, hereafter the service the OM/RR-protocol offers is termed the OM-service. The OM-service includes the RR-service and basically enables two different *Managed Objects*, i.e. two service users, to communicate with one-another when they have established an *association*, using the OM-service.

Usually, a distinction can be made between binary and multiple associations. This distinction will have its impact in the dynamic specification of the OM-service. As was understood within CMG, and was hinted at in [AVV97], an association involved only two Managed Objects, and as such associations are hereafter considered binary. The definition for an association, used in this report is as follows:

Definition 2.2 (Association) *A relation between two different Managed Objects and the OM-service provider. This relation can be* enabled *(established) for the use of communication and* disabled *(relinquished).*
Each association is uniquely identifiable by its two Managed Objects.

The term *session* is used to denote the period between the successfully establishing and the subsequent relinquishing of an association. Figure 2.1 shows the relation between the terms *enabled*, *disabled* and *session* for an association against a time-axis.
When two Managed Objects start communicating, there is always one Managed Object that initiated the establishing of the association both use. Often, this initiative plays a role during the lifetime of the association, and as such, it is easy to make a distinction between the Managed Object initiating the association and the Managed Object responding to the initiation. In [AVV97],

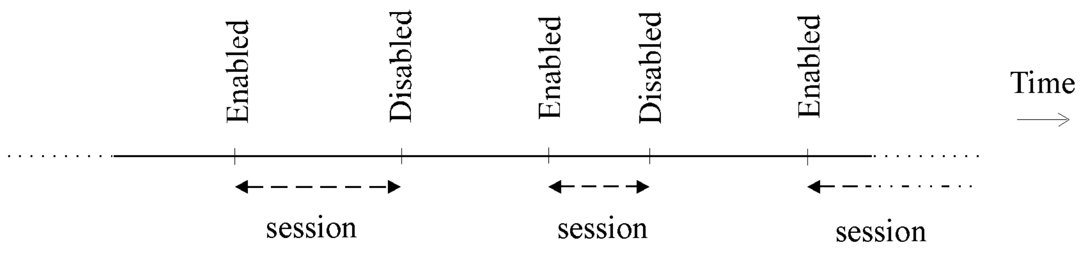

Figure 2.1: *A snapshot of three sessions.*

both the terms *Invoker* and *Performer*, and *Initiator* and *Responder* are coined, yet the relation between them is unclear. Because of this uncertainty, it has been decided to use neither terms, but introduce new terms that will be defined uniquely. In accordance to [Kap91], henceforth the term *Manager* is used for the Managed Object initiating the association and the term *Agent* is used for the Managed Object responding to the initiation.

The environment of the OM-service is depicted in figure 2.2.

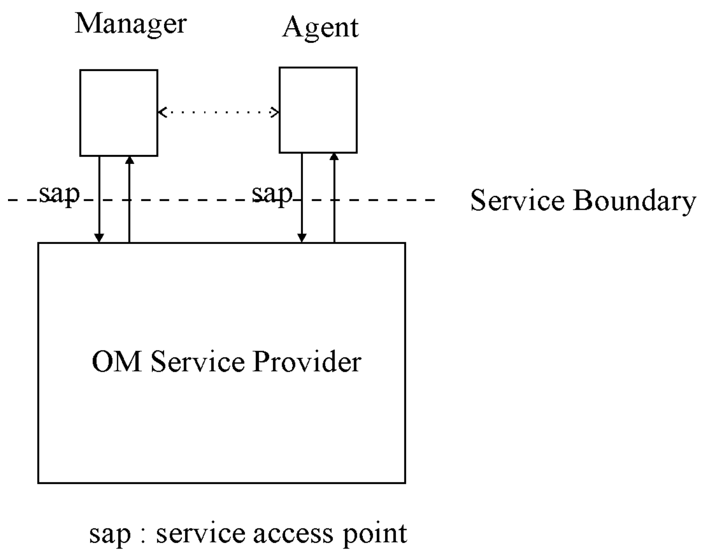

Figure 2.2: *The OM-service regarded as a black box.*
Both Manager and Agent are OM-service users.

The protocol stack used in this thesis (see figure 2.3) differs from the one given in [AVV97] (see figure 2.4). At least three reasons exist for this difference:

- No clear relationship was defined between the following three protocol entities: the OM-protocol entity, the RR-protocol entity and the AM-protocol entity (the protocol entity used to establish and relinquish associations).
- It was unclear how the AM protocol entity itself was layered.
- The service the AM-protocol entity has to offer was not clearly specified.

The difference between the old stack and the new stack is evident: the AM-protocol entity is erased and its services (as far as they could be identified) are taken over by the OM-protocol entity. This layering different from the original stack is justified by the fact that binary associations are considered at the OM-service level; these associations are regarded as an integral part of the OM-service. Furthermore, the protocol stack as depicted in figure 2.3, is how the original protocol stack was interpreted within CMG.

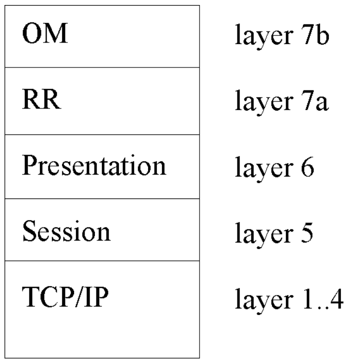

Figure 2.3: *The protocol stack used in this report*

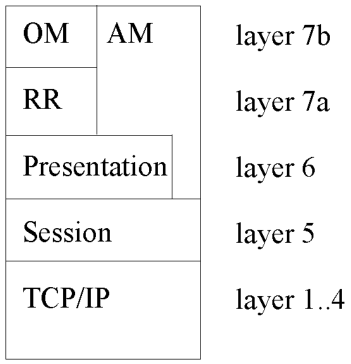

Figure 2.4: *The protocol stack according to AVV*

The service which the fifth and sixth layer of the OSI-model — the session and the presentation layer — are supposed to offer is only informally described in a few sentences in [AVV97]. The presentation layer is described as being able to code and decode PDUs and the session layer is described to use “sockets”. No specification of these layers could be found, and this is thus considered to be an omission of the OM/RR-protocol specifications.

2.2 Determining the Service Elements and Service Primitives for the OM-service

This Section deals with the service elements described in [AVV97]. In [Kap91], as well as in [AVV97] a distinction is made between two types of service the OM-service offers to its users:

- a Management Operation Service (MOS).
- the Management Notification Service (MNS).

This distinction was introduced because the service elements belonging to them were considered to be orthogonal from the point of view of management: the MOS described *operations*, e.g. actions that have to be performed, and the MNS merely describes reports and notifications on these reports. Besides this difference in the optic of management, there is a difference in the initiation of the MOS and the MNS. The Manager is allowed to initiate both MOS and MNS, whereas the Agent is only allowed to initiate the MNS. This is not explicitly stated in [AVV97], but has been assumed, as this was the case in [Kap91].
The following Sections will discuss the MOS and the MNS in greater detail.

2.2.1 The MOS

The service provided by the MOS is used to manage and control objects. The service elements that can be distinguished, according to [AVV97] are the following:

get, set, action, create, delete.

Furthermore, according to [AVV97], the following distinction can be made:

- The get, set, action, create and delete service elements are all user confirmed service elements.
- The set service element can also be an unconfirmed service element.

These service elements can be parameterized with specific operations, both Manager and Agent have agreed upon. In this thesis, the MOS shall be extended with two more service elements: the *enable* and the *disable* service elements, which, according to [AVV97] are parameters for the action service elements. A time-sequence diagram in [AVV97] (pages 56-57), however, describes a provider confirmed service for these specific parameters, instead of a user confirmed service. Since this clearly is different with respect to the observable behaviour, and shall have its impact on the service specification, the enable and disable parameters have been introduced as new service elements. These service elements are provider confirmed, and this is done in accordance with [Kap91].
In order to be able to distinguish between the two set service elements, the user confirmed set service element is hereafter refered to as the *cset* service element, whereas the unconfirmed set service element is henceforth refered to as the *uset* service element.
The set of service elements providing the MOS now is the following:

get, cset, uset, action, create, delete, enable, disable.

These service elements have different meanings as to what purpose they are used for. These purposes are described in [AVV97] as follows:

- A Manager issues a get in order to obtain the attributes of the Agent.
- A Manager uses a cset, to set the attributes of the Agent and information concerning the operation is returned to the Manager.
- A Manager uses a uset, to set the attributes of the Agent. No information is returned to the Manager.
- A Manager issues an action to have the Agent perform this action and return the results hereof.
- A Manager uses the create service element to have the Agent create yet another (non-existing) Managed Object and return the results of the operation.
- A Manager uses the delete service element to have the Agent delete another (existing) Managed Object and return the results of the operation.
- A Managed Object uses the enable to establish an association with a remote Managed Object and return the results of the operation.
- A Managed Object uses the disable to relinquish an association with a remote Managed Object and return the results of the operation.

The set of service elements can be described using atomic service primitives. In this thesis, the following convention is used: the service element's name is extended with

- a *req* for the request of a service element by the initiator of this same service element.
- an *ind* for the indication of a service element.
- a *res* for the response to a previously communicated indication.

Service Element	Service Primitive	type of service
enable	enable.req enable.ind enable.conf	provider confirmed
disable	disable.req disable.ind disable.conf	provider confirmed
get	get.req get.ind get.res get.conf	user confirmed
set	cset.req cset.ind cset.res cset.conf	user confirmed
create	create.req create.ind create.res create.conf	user confirmed
delete	delete.req delete.ind delete.res delete.conf	user confirmed
action	action.req action.ind action.res action.conf	user confirmed
set	uset.req uset.ind	unconfirmed

Table 2.1: *The service elements and service primitives of the MOS*

- a *conf* for the confirmation of a previously communicated request of a service element.

Table 2.1 shows the relationship between the service primitives and the corresponding service element, and the kind of service that is offered. Note that this table does not specify any dynamic behaviour.

The [AVV97] document is very vague about the effects of the service elements on the behaviour of the OM/RR-protocol itself. Apart from the general lack of a dynamic service specification (e.g. relations between a get.req and a get.ind), the following information concerning the create and delete service elements could not be found:

- The implications of these operations are not specified. It is for instance unclear how the operation affects the Manager and Agent that perform the operation, or how it affects the Managed Objects that have an association with the Managed Object that is to be deleted.
- The mechanism the Agent must use to create or delete a Managed Object is not specified, and as such, its effects are unclear.
- The circumstances under which these service elements can be used are not described.

In a conversation with AVV, it was found out that the action, create, delete and the user confirmed set service elements have the same implications on the OM-service as the get service element. It was therefore decided that these additional service elements are not included in the OM-service

Service Element	Service Primitive	type of service
enable	enable.req enable.ind enable.conf	provider confirmed
disable	disable.req disable.ind disable.conf	provider confirmed
get	get.req get.ind get.res get.conf	user confirmed
set	uset.req uset.ind	unconfirmed

Table 2.2: *The MOS in the OM-service model*

model described in this thesis, as no extra conclusions can be drawn from the inclusion of these service elements. The total number of MOS service elements, and as a consequence, the number of service primitives belonging to the MOS is reduced. This reduction is necessary to create LTSs as small as possible, in order to perform subsequent validation on these models using EUCALYPTUS (see Chapter 5).
Table 2.2 shows the relation between the service elements and the service primitives that are offered in the resulting OM-service model.

2.2.2 The MNS

The MNS is a service that is provided in order to enable a Managed Object (in a role of an Agent) to send reports to its peer (i.e. a Manager) about its own status, using a previously established association.

The [AVV97] document describes four kinds of *report* service elements. The impact these reports have on the service provider is not clear. Therefore, for the time being, it is assumed there is no impact. As a result, these four different report service elements are mapped onto a generic report service element, that represents all four kinds of report service elements. Furthermore the [AVV97] document states that these reports can either be classified as *addressed* or *non-addressed*. Finally, a *notification* service element is introduced. This notification is used by Managers to notify other Managed Objects in the role of Managers, that a report received earlier has been read. This notification service element is non-addressed.
The total set of service elements then is the following set:

addressed_report, non_addressed_report, non_addressed_notification.

Figure 2.5 shows the results of the sending of a non-addressed report by a Managed Object in a role of an Agent to Managed Objects in the role of Managers. Note that some form of broadcast is suggested.

In [AVV97] the term *subscription* is coined. This term is used to distinguish between Managers which are capable of receiving non-addressed reports from their Agent, and Managers which are capable of receiving addressed reports *only*. Furthermore, notifications are sent by one Manager to *all other* Managers, subscribed to this Agent (note that one Managed Object can be Agent in more than one association, only then the Managers are different Managed Objects!).
An addressed report is sent by a Managed Object (in the role of an Agent) to exactly one peer,

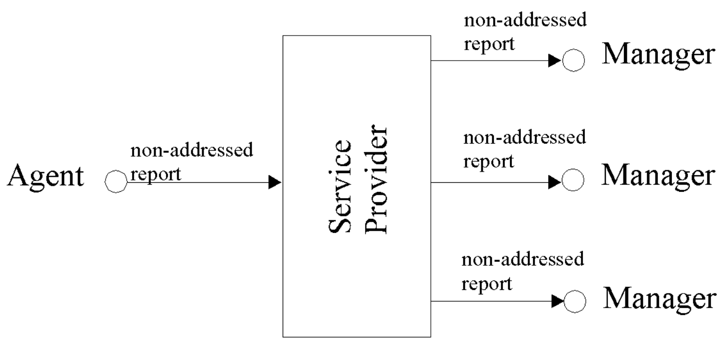

Figure 2.5: *The use of the non-addressed report service*

whilst the non-addressed reports are sent to its subscribers. In order to enable the subscribers to inform the other subscribed Managed Objects, a non-addressed notification is used.

Here a problem emerges: the term subscription is not defined in [AVV97] (nor in any other document). Due to this gap no proper relationship between the users of the non-addressed report and notification service can be described. At CMG it was believed, the subscriptions could be characterized as follows:

Definition 2.3 (Subscription) *A Managed Object in the role of a Manager has a* subscription *to a Managed Object in the role of an Agent if the following two conditions are both met:*

1. *An association is established between the Manager and the Agent.*
2. *The association that exists between the Manager and the Agent* permits *non-addressed reports and notifications concerning these reports, to be sent to the Manager. Moreover, this "permit", is assumed to be something that can be verified.*

This definition however, is too vague to be used in a model. For instance, the permission mechanism should be defined.

Furthermore, the dynamic behaviour of the non-addressed report service is not described in [AVV97]. It is unclear whether it acts either as the "addressed" report service (in this case the Agent is responsible for distributing the non-addressed reports to the Managers), or as a "broadcast" report service (in this case the OM-service is responsible for distributing the non-addressed reports to the subscribed Managers).
The former can be sufficiently described by the "normal" report service element, while the latter poses some problems, which shall be explained below:

In order to support the non-addressed report service (in the latter case), at least a multiple association is needed (actually a one-to-many association): the OM-service must allow a Managed Object (sender) to distribute *one* report to a set of Managed Objects (receivers). Furthermore, the sender is not interested in who the receivers are, since no addresses are used. It follows that the OM-service must somehow be able to access the information concerning these receivers in order to deliver the report to them.

The use of multiple associations, however, is not in line with both [AVV97] and [Kap91], since in both documents a binary association is suggested.

As a consequence, the non-addressed -report and -notification service, under the assumption that they cannot be described by the addressed services, cannot be supported. This can be mended by

Service Element	Service Primitive	type of service
report	report.req report.ind	unconfirmed
notification	notification.req notification.ind	unconfirmed

Table 2.3: *The service elements and service primitives of the MNS*

allowing a layer on top of the OM-layer (i.e. the service user) to take over this service, which would result in a selective broadcast over the set of associations. Another solution could be to describe multiple associations instead of the binary association, but this would no longer be in accordance with [AVV97]. In this document, the first solution shall be adopted, since no guidelines can be found in [AVV97] concerning this matter.
Two consequences follow from adopting this solution:

- The non-addressed notification service element is replaced with an addressed notification service element that uses only one association.
- The OM-service must allow for both the Manager and the Agent to initiate the notification service.

This latter consequence is the result of the fact that the Agent is the only Managed Object that has knowledge about *all* its subscribers, and can therefore be the only Managed Object, that can pass on notifications, sent by one Manager, to the other subscribers.

Another issue is, that, strange as it may seem, no causality between reports and notifications is described in [AVV97], thus allowing for the following "anomaly":

Suppose Agent A has two subscribers, Managers B and C. Now a non-addressed report is sent (either by the responsibility of the OM-service or the Agent), which arrives at C much later than at B. In this case, B could already have sent a notification to A, which A will pass on to C, and can therefore arive *before* the corresponding report at C (i.e. C receives a notification on a report not (yet) received).

This anomaly is due to the fact that a message reordering system is considered (see page 20).

In light of the problems mentioned above, the MNS that is offered has slightly been changed. The non-addressed service elements cannot be offered, and as such, the complete set of service elements belonging to the MNS is the following:

report, notification

Clearly, both MNS service elements are unconfirmed service elements, as described in [AVV97]. Table 2.3 lists the service elements and the corresponding service primitives offered in the MNS as it will be modelled.

In the model under consideration in this report, the MNS is further restricted to only one service element: the report service element. This is done for reasons of conciseness. The notification service element, when initiated by a Manager, performs the same as the unconfirmed MOS uset service element; a notification service element, initiated by an Agent, acts the same as the report service element.

Table 2.4 shows the service element and the service primitives for the MNS under consideration in the model described in this thesis.

Service Element	Service Primitive	type of service
report	report.req report.ind	unconfirmed

Table 2.4: *The MNS in the OM-service model*

2.3 The Dynamic Specification of the OM-service Model

In this Section, the dynamic behaviour of the OM-service model shall be described, using the FDT LOTOS. The specification shall include the full Abstract Data Types. Furthermore, a constraint-oriented approach is used to formulate the requirements on the OM-service.
The complete service boundary is represented by the single gate `OSAP` (short for *OM Service Access Point*). Due to a problem with CÆSAR.ADT (discussed in Section 2.3.1, page 16), the polymorphic gate features (i.e. the possibility of having more than one distinct event structure at a gate) of LOTOS shall be used. There are two distinct event structures at gate `OSAP` described as follows:

```
  OSAP <var: AI> <var: OSP> <var: AId>
  OSAP <var: AI> <var: OSP> <var: AId> <var: BOOL>
```

The sort `AI` (short for *Address Identifier*) is used to identify the Managed Object that is addressed in a communication over gate `OSAP`, the sort `OSP` (short for *OM Service Primitive*) is used to identify the service primitive that is communicated over gate `OSAP` and the sort `AId` (short for *Association Identifier*) is used to distinguish between possible different associations. Finally, the sort `BOOL`, which defines the booleans, is used in cases where the service primitive communicated requires a boolean.
The specification of the OM-service model is that of a never terminating one, written as follows:

```
SPECIFICATION OM_Service[OSAP] : NOEXIT
```

This specification uses the standard libraries `boolean`, specifying the sort `bool` and the most frequently used functions, and `naturalnumber`, specifying the sort `nat` and the most frequent used functions.

```
LIBRARY
  boolean,
  naturalnumber
ENDLIB
```

The Abstract Data Types, used in the descriptions in Section 2.3.2, are specified in Section 2.3.1.

2.3.1 The Abstract Data Types used in the OM-service Model

The datatypes and functions used in the OM-service model are written using the static part of LOTOS: the Abstract Data Types (ADTs). These ADTs are specified in the language ACT-ONE. Several adaptations have been made to allow the toolbox EUCALYPTUS, especially the tool CÆSAR.ADT, to deal with the ADTs, such as the addition of special comments like `(*! constructor *)` and `(*! implementedby ... constructor *)`. These constructions are necessary for CÆSAR.ADT to recognise which elements are *sort constructors* and which are not. The `(*! implementedby ... *)` construction tells CÆSAR.ADT which C name should be used in the translation from ACT-ONE to C, and is not mandatory. In order to generate a *Labeled Transition System* for a LOTOS-specification, only enumerable sorts can be used (i.e. a sort for which a mapping to a subset of the natural numbers exists). Note that the language used to specify the ADTs, ACT-ONE, is not case-sensitive.

The first ADT that is introduced is the type `Address_Identifier`, specifying the sort `AI`. This sort is used to denote the address of a SAP and can be used to identify the Managed Object that is connected to that SAP. In this model, the sort `AI` consists only of two elements, because only two Managed Objects are considered (see also the restrictions discussed in Section 2.3.2 and Appendix A).

```
TYPE Address_Identifier

IS Boolean
SORTS AI
OPNS offset    (*! constructor *)        : -> AI
     neighbour (*! constructor *)        : -> AI
     _eq_                                : AI, AI -> Bool
EQNS
OFSORT Bool
       offset eq offset                   = true;
       offset eq neighbour                = false;
       neighbour eq offset                = false;
       neighbour eq neighbour             = true;

ENDTYPE (* Address_Identifier *)
```

Apart from the above specified type, the type `Association_Identifier` is specified, describing the sort `AId`, the elements of which uniquely identify each possible association. This is done by associating the addresses of the Manager and the Agent to the association identifier, since this combination is unique in every association. The type `AId` is specified below:

```
TYPE Association_Identifier

IS Address_Identifier, Boolean
SORTS AId
OPNS
  AId        (*! constructor *)         : AI, AI -> AId
  Initiator                             : AId -> AI
  Responder                             : AId -> AI
  _eq_       (*! implementedby eq_aid *) : AId, AId -> Bool
  _ne_       (*! implementedby ne_aid *) : AId, AId -> Bool
EQNS
FORALL a1,a2 : AI, as1,as2 : AId
OFSORT AI
       Initiator(AId(a1,a2))         = a1;
       Responder(AId(a1,a2))         = a2;
OFSORT Bool
       as1 eq as2  = (Initiator(as1) eq Initiator(as2)) and
                     (Responder(as1) eq Responder(as2));
       as1 ne as2  = not(as1 eq as2);

ENDTYPE (* Association_Identifier *)
```

Finally, the ADT `OM_Service_Primitives` is introduced to distinguish between the different service primitives. This type was adapted to the needs of CÆSAR.ADT, and thus was significantly simplified (see also Appendix A for the original type specification).

```
TYPE OM_Service_Primitives

IS
SORTS OSP
OPNS usetreq         (*! constructor *)  :  -> OSP
```

```
     enablereq       (*! constructor *)  :  -> OSP
     disablereq      (*! constructor *)  :  -> OSP
     reportreq       (*! constructor *)  :  -> OSP
     getreq          (*! constructor *)  :  -> OSP
     usetind         (*! constructor *)  :  -> OSP
     enableind       (*! constructor *)  :  -> OSP
     disableind      (*! constructor *)  :  -> OSP
     reportind       (*! constructor *)  :  -> OSP
     getind          (*! constructor *)  :  -> OSP
     getres          (*! constructor *)  :  -> OSP
     enableconf      (*! constructor *)  :  -> OSP
     disableconf     (*! constructor *)  :  -> OSP
     getconf         (*! constructor *)  :  -> OSP

ENDTYPE(* OM_Service_Primitives*)
```

This type simply defines another enumerated type. The sort `OSP`, defined in the Appendix A, was defined in an ADT with the following signature:

```
...
SORTS OSP'
OPNS enablereq   (*! constructor *) : AID, ID -> OSP'
     enableind   (*! constructor *) : AID, ID -> OSP'
     enableconf  (*! constructor *) : AID, ID, Bool -> OSP'
...
```

When trying to translate an ADT with the above sketched signature, CÆSAR reports warnings about the theoretical limitation stating that no enumeration of this type definition could be made, even when the sorts `AID, ID` and `Bool` are finite. Since an enumeration does exist for this ADT (the number of elements of the sort `OSP'` is clearly finite), this is considered an omission of CÆSAR, as the construction sketched above is useful at times.

In order to have an easy distinction between two service primitives, equality is necessary. One way to achieve this is by enumerating this sort by defining a mapping onto a subset of natural numbers. For the sake of readability, an auxiliary ADT defining a sort that contains sixteen elements representing the first sixteen positive natural numbers is introduced. This ADT is the type `SixteenTuplet`.

```
TYPE SixteenTuplet

IS Boolean, NaturalNumber
SORTS Tuplet
OPNS One       (*! implementedby one constructor *)      : -> tuplet
     Two       (*! implementedby two constructor *)      : -> tuplet
     Three     (*! implementedby three constructor *)    : -> tuplet
     Four      (*! implementedby four constructor *)     : -> tuplet
     Five      (*! implementedby five constructor *)     : -> tuplet
     Six       (*! implementedby six constructor *)      : -> tuplet
     Seven     (*! implementedby seven constructor *)    : -> tuplet
     Eight     (*! implementedby eight constructor *)    : -> tuplet
     Nine      (*! implementedby nine constructor *)     : -> tuplet
     Ten       (*! implementedby ten constructor *)      : -> tuplet
     Eleven    (*! implementedby eleven constructor *)   : -> tuplet
     Twelve    (*! implementedby twelve constructor *)   : -> tuplet
     Thirteen  (*! implementedby thirteen constructor *): -> tuplet
     Fourteen  (*! implementedby fourteen constructor *): -> tuplet
     Fifteen   (*! implementedby fifteen constructor *) : -> tuplet
     Sixteen   (*! implementedby sixteen constructor *) : -> tuplet
     _eq_ (*! implementedby eq_tuplet *) : tuplet, tuplet -> Bool
     _ne_ (*! implementedby ne_tuplet *) : tuplet, tuplet -> Bool
     h    (*! implementedby h_tuplet *)  : tuplet -> Nat
EQNS
FORALL x,y : Tuplet
OFSORT Bool
       x eq y    = h(x) eq h(y);
       x ne y    = h(x) ne h(y);
OFSORT Nat
       h(One)        = 0;
       h(Two)        = succ(h(One));
       h(Three)      = succ(h(Two));
       h(Four)       = succ(h(Three));
       h(Five)       = succ(h(Four));
       h(Six)        = succ(h(Five));
       h(Seven)      = succ(h(Six));
       h(Eight)      = succ(h(Seven));
       h(Nine)       = succ(h(Eight));
       h(Ten)        = succ(h(Nine));
       h(Eleven)     = succ(h(Ten));
       h(Twelve)     = succ(h(Eleven));
       h(Thirteen)   = succ(h(Twelve));
       h(Fourteen)   = succ(h(Thirteen));
       h(Fifteen)    = succ(h(Fourteen));
       h(Sixteen)    = succ(h(Fifteen));

ENDTYPE (* SixteenTuplet *)
```

The abovedefined set is now used to map the Service Primitives onto the natural numbers and functions are introduced that define the identification of a service primitive as follows. Note that the enumeration that function `Map` defines could have been specified without the help of the ADT `SixteenTuplet`.

```
TYPE OSP_Classifier

IS SixteenTuplet, OM_Service_Primitives
OPNS Map               : OSP -> Tuplet
     IsUsetReq         : OSP -> Bool
     IsEnableReq       : OSP -> Bool
     IsDisableReq      : OSP -> Bool
     IsReportReq       : OSP -> Bool
     IsGetReq          : OSP -> Bool
     IsUsetInd         : OSP -> Bool
     IsEnableInd       : OSP -> Bool
     IsDisableInd      : OSP -> Bool
     IsReportInd       : OSP -> Bool
     IsGetInd          : OSP -> Bool
     IsGetRes          : OSP -> Bool
     IsEnableConf      : OSP -> Bool
     IsDisableConf     : OSP -> Bool
     IsGetConf         : OSP -> Bool
EQNS
FORALL prim : OSP
OFSORT Tuplet
       Map(USetReq)              = One;
       Map(EnableReq)            = Two;
       Map(DisableReq)           = Three;
       Map(ReportReq)            = Four;
       Map(GetReq)               = Five;
       Map(USetInd)             = Six;
       Map(EnableInd)           = Seven;
       Map(DisableInd)          = Eight;
       Map(ReportInd)           = Nine;
       Map(GetInd)              = Ten;
       Map(GetRes)              = Eleven;
       Map(EnableConf)          = Twelve;
       Map(DisableConf)         = Thirteen;
       Map(GetConf)             = Fourteen;
OFSORT Bool
       IsUsetReq(prim)                  = map(prim) eq One;
       IsEnableReq(prim)                = map(prim) eq Two;
       IsDisableReq(prim)               = map(prim) eq Three;
       IsReportReq(prim)                = map(prim) eq Four;
       IsGetReq(prim)                   = map(prim) eq Five;
       IsUsetInd(prim)                  = map(prim) eq Six;
       IsEnableInd(prim)                = map(prim) eq Seven;
       IsDisableInd(prim)               = map(prim) eq Eight;
       IsReportInd(prim)                = map(prim) eq Nine;
       IsGetInd(prim)                   = map(prim) eq Ten;
       IsGetRes(prim)                   = map(prim) eq Eleven;
       IsEnableConf(prim)               = map(prim) eq Twelve;
       IsDisableConf(prim)              = map(prim) eq Thirteen;
       IsGetConf(prim)                  = map(prim) eq Fourteen;

ENDTYPE (* OSP_Classifier *)
```

A distinction towards different classes of OSP

Having defined the set of OM service primitives, the foundation is there to introduce some user-friendly ADTs that shall be used in the process descriptions given in subsequent Sections and Chapters. Several classes of OSPs can be distinguished, each characterized by some common property. The most intuitive distinction that can be made is the one classifying each service primitive in either the class of *requests*, *indications*, *responses* or *confirmations*. To this end, the following ADT is introduced.

```
TYPE OSP_Servicetype

IS OSP_Classifier
OPNS IsReq                : OSP -> Bool
     IsInd                : OSP -> Bool
     IsRes                : OSP -> Bool
     IsConf               : OSP -> Bool
EQNS
FORALL prim : OSP
OFSORT Bool
       IsReq(prim)          = IsUSetReq(prim) or IsEnableReq(prim) or
                              IsDisableReq(prim) or IsGetReq(prim) or
                              IsReportReq(prim);
       IsInd(prim)          = IsUSetInd(prim) or IsEnableInd(prim) or
                              IsDisableInd(prim) or IsGetInd(prim)  or
                              IsReportInd(prim);
       IsRes(prim)          = IsGetRes(prim);
       IsConf(prim)         = IsEnableConf(prim) or IsDisableConf(prim) or
                              IsGetConf(prim);
ENDTYPE(*OSP_ServiceType*)
```

Besides the abovedefined classification, another classification is evident, namely the one that classifies the Service Primitives as Service Elements :

```
TYPE OSP_Elements is OSP_Classifier, Boolean

OPNS IsUSet            : OSP -> Bool
     IsEnable          : OSP -> Bool
     IsDisable         : OSP -> Bool
     IsReport          : OSP -> Bool
     IsGet             : OSP -> Bool
EQNS
FORALL prim : OSP
OFSORT bool
       IsUSet(prim)        = IsUSetReq(prim) or IsUSetInd(prim);
       IsEnable(prim)      = IsEnableReq(prim) or IsEnableInd(prim) or
                             IsEnableConf(prim);
       IsDisable(prim)     = IsDisableReq(prim) or IsDisableInd(prim) or
                             IsDisableConf(prim);
       IsReport(prim)      = IsReportReq(prim) or IsReportInd(prim);
       IsGet(prim)         = IsGetReq(prim) or IsGetInd(prim) or
                             IsGetRes(prim) or IsGetConf(prim);

ENDTYPE (* OSP_Elements *)
```

The third class that can be distinguished is the class that describes whether a Service Primitive is an *unconfirmed*, *userconfirmed* or a *providerconfirmed* service. The following ADT defines this class:

```
TYPE OSP_Type

IS Boolean, OSP_Elements
OPNS IsUnConf          : OSP -> Bool
     IsUserConf        : OSP -> Bool
     IsProviderConf    : OSP -> Bool
EQNS
FORALL prim : OSP
OFSORT Bool
       IsUnConf(prim)        = IsUSet(prim) or IsReport(prim);
       IsUserConf(prim)      = IsGet(prim);
       IsProviderConf(prim)  = IsEnable(prim) or IsDisable(prim);

ENDTYPE (* OSP_Type *)
```

This concludes the ADT specifications for the OM-service model.

2.3.2 The Dynamic Behaviour of the OM-service Model

The behaviour of the OM-service model is based on several requirements. Some of these requirements have been assumed, others can be found (implicitly or explicitly) in [AVV97]. The list below describes these requirements:

1. An Association can be identified by its Manager and its Agent (thus the tuple (Manager,Agent) uniquely identifies the association) — This has been assumed.
2. Once an association is established, it can only be relinquished by a disable request. This disable request can be issued both by the Manager and by the Agent and always succeeds — [AVV97].
3. A service provider initiated termination is allowed — assumed.
4. Associations can only be set up between existing, different Managed Objects — assumed.

5. Both the disable indication and the disable confirmation denote the definite end of the corresponding association. — [AVV97]
6. Message reordering is allowed. — [AVV97]

These requirements are stated in the following top level process:

```
BEHAVIOUR

  association[OSAP] (AID(offset,neighbour))

WHERE
```

One of the requirements omitted in this OM-service model is the possibility of having a number of concurrent associations. The OM-service model describes only *one* association between *two* different OM-protocol entities. Modelling a (possibly infinite) number of associations can easily be described by the following process:

```
PROCESS associations[OSAP] (A : Set_Of_AId) : NOEXIT :=

  CHOICE aa : AId [] [aa isin A] ->
  (association[OSAP] ||| i; associations [OSAP] (remove (aa, A) ) )

ENDPROC (* associations *)
```

where the set `A` contains the set of all possible associations. However, restrictions imposed by CÆSAR do not allow for such (possibly unbounded) recursive process instantiation, and as such process `associations` cannot be used. In the OM-service model no concurrent associations are considered to overcome this problem. This is justified by the fact that the associations are considered to be binary and hence are believed not to interfere with each-other. Another reason for considering only one association at a time is the purpose of the specification: using CÆSAR, a *Labeled Transition System* (LTS) is to be built (see Chapter 5), which easily can reach several millions of states and transitions if for example concurrent associations are modelled.

The decomposition of one association

One association is composed of a LOTOS-process representing the local constraints for the Manager side of the association (analogously called process `manager`) and a LOTOS-process representing the local constraints for the Agent side of the association (analogously called process `agent`). These processes are completely independent of each other, and can therefore be modelled using the LOTOS interleaving (`|||`) operator. The end-to-end constraints are specified in process `sync`. This process is used to relate the events in process `manager` to events in process `agent` using the LOTOS parallel (`||`) operator. Another simplification that is made is the omission of several dataparameters in the communication of service primitives. Data, not having any impact on the OM-service is omitted for conciseness' sake. Time is one such parameter.

```
  PROCESS association [OSAP] (aa : aid) : NOEXIT :=

    (agent [OSAP] (aa)  ||| manager [OSAP] (aa) ) || sync [OSAP] (aa)

  ENDPROC (* association *)
```

Subsequently, processes `agent`, `manager` and `sync` shall be described. First the local end constraints shall be specified. The end-to-end constraints descriptions can be found on page 26 and onward.

The Local End Constraints

The processes **manager** and **agent** specify four stages which reflect the view their real-life counterpart Managed Objects have on the state of the association under consideration:

1. The association is in its setup-time. The enable service element is used in this state to establish the association. This means that no Managed Object can use the association to send data, but that the association is still "under construction". This state is marked as the *Initial* state.

2. Should the setup of the association be successful, the next phase is the data transfer phase. In this phase, both the Manager and the Agent can send and receive data using this association. Message reordering is allowed in this state. This state is marked as the *Communication* state.

3. The association is released. In this case, the end of the communication is marked by a disable request by one of the Managed Objects. After this request is issued, only confirmations on previously emitted requests can be received. This state is marked as the *Release* state.

4. The association is aborting. This means that an external event, initiated by the peer Managed Object or by the Service Provider, has forced the association to terminate. Communication is immediately blocked, and the association is no longer operable. This state is marked as the *Aborting* state.

The abovementioned states are, together with the transitions, depicted in figure 2.6. As stated before, these states reflect the views the Manager and Agent have on the association, but it is not the state of the association itself, since timing differences can complicate combining the states observed by the Manager and the Agent into one state for the association as a whole. Besides, this is one of the fundamental problems in distributed systems.

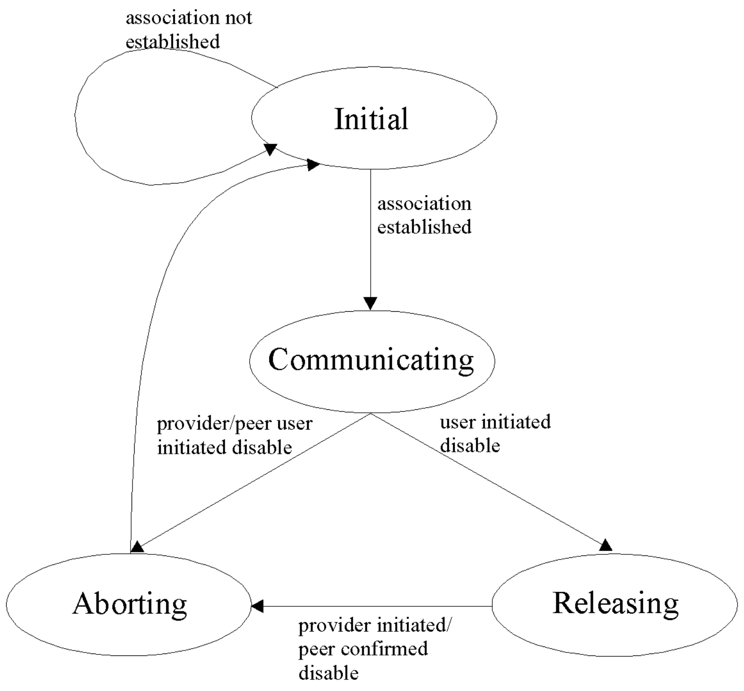
association not established
Initial
association established
Communicating
provider/peer user initiated disable
user initiated disable
Aborting
Releasing
provider initiated/ peer confirmed disable

Figure 2.6: *The Managed Object's view of an Association*

This statediagram allows easy translation into a constraint oriented description for the processes **manager** and **agent** as follows:

```
PROCESS manager[OSAP](aa : aid) : NOEXIT :=

   minitiate [OSAP] (aa)
  >>
  (((mcommunicate [OSAP] (aa) || initiaterelease [OSAP] (initiator(aa)) )
    [>
     forcerelease [OSAP] (initiator(aa))  )
  ||
    syncdisable [OSAP] (initiator(aa)) )
  >>
    manager [OSAP] (aa)

ENDPROC (*manager*)
```

Some remarks are in order: process **minitiate** specifies the initial state for the manager, process **mcommunicate** specifies the communicating state, and the synchronizing operator (||) is used to restrict communication when a user initiated disable occurs (i.e. when a transition to the releasing state occurs). Process **forcerelease** is used to shift to the aborting state, and process **syncdisable** specifies the local ordering on the disable service primitives. Analogously, process **agent** is specified below:

```
PROCESS agent[OSAP](aa : aid) : NOEXIT :=

   ainitiate [OSAP] (aa)
   >>
   (((acommunicate [OSAP] (aa) || initiaterelease [OSAP] (responder(aa)) )
    [>
     forcerelease [OSAP] (responder(aa))  )
  ||
    syncdisable [OSAP] (responder(aa)) )
  >>
    agent [OSAP] (aa)

ENDPROC (*agent*)
```

The Initial State

The processes **minitiate** and **ainitiate** handle the different aspects of initializing an association. They specify which service primitives can (locally) occur in which order. If and only if setting up the association succeeds, the communication of data can start. Since the enable service-element is a provider confirmed service-element, the Agent is offered an association with a Manager, and has no means of disallowing this association to be set up. The OM-service provider, however, may or may not decide to establish the association. In the case that the service provider disallows the establishment of the association, the Agent is not offered an enable indication, whereas when the service provider does allow the association to be established, the Agent is offered the indication. However, this is specified in the end-to-end constraints.
The processes are specified as follows :

```
PROCESS minitiate [OSAP] (aa : aid) : EXIT:=

  OSAP !initiator(aa) !enablereq !aa;
  OSAP !initiator(aa) !enableconf !aa ?b : bool;
  ([not(b)] -> minitiate [OSAP] (aa)
  []
   [b] -> EXIT )
```

```
ENDPROC (*minitiate*)


PROCESS ainitiate [OSAP] (aa : aid) : EXIT :=

  OSAP !responder(aa) !enableind !aa;
  EXIT

ENDPROC (*ainitiate*)
```

The Communicating State

The state that can be reached from the initial state, is the communicating state. In this state, all data transfer is done. Corresponding to the service description, a distinction is made between the MOS and the MNS, and this reflects in the process descriptions. The assumption is made, that in this phase, the Service Provider may reorder the messages. Actually in this state, an infinite number of communications can occur, but this is modelled using just one user confirmed get service element, one unconfirmed set service element and an unconfirmed report service element. The same reasons as before in process `associations` apply for these simplifications (i.e. the recursive process instantiation). Furthermore, the reordering of an infinite number of messages in this state is not considered to cause more problems than the reordering of three messages. The processes `acommunicate` and `mcommunicate` are both specified below :

```
PROCESS mcommunicate[OSAP](aa : aid) : NOEXIT :=

  mnsm[Osap](initiator(aa)) ||| mosm[Osap](initiator(aa))

ENDPROC (* mcommunicate *)

PROCESS acommunicate[Osap](aa : aid) : NOEXIT :=

  mnsa[Osap](responder(aa)) ||| mosa[Osap](responder(aa))

ENDPROC (* acommunicate *)
```

The processes `mosm` and `mosa` make a distinction with respect to the classes of service-types (i.e. user confirmed, provider confirmed and unconfirmed). Since pure interleaving is used, message reordering among these service-types is still possible.
The processes are then straightforward, and introduced without further explanation :

```
PROCESS mosm[OSAP](x : ai) : NOEXIT :=

  OSAP !x !disablereq ?aa : AID ;
  STOP
|||
  OSAP !x !usetreq ?aa :AID ;
  STOP
|||
  OSAP !x !getreq ?aa : AID ;
  OSAP !x !getconf ?aa : AID ;
  STOP

ENDPROC (* mosm *)


PROCESS mosa[OSAP](x : ai) : NOEXIT :=
    OSAP !x !disablereq ?aa : AID ;
    STOP
  |||
    OSAP !x !usetind ?aa : AID ;
    STOP
  |||
    OSAP !x !getind ?aa : AID ;
    OSAP !x !getres ?aa : AID ;
    STOP

  ENDPROC (* mosa *)
```

The processes describing the MNS for both the Manager and the Agent are described below. Here, also the restriction is that only one message is considered, being the report service element:

```
  PROCESS mnsm[OSAP](x : ai) : NOEXIT :=

    OSAP !x !reportind ?aa : AID ;
    STOP

  ENDPROC (* mnsm *)


  PROCESS mnsa[OSAP](x : ai) : NOEXIT :=

    OSAP !x !reportreq ?aa : AID ;
    STOP

  ENDPROC (* mnsa *)
```

The Releasing State

The assumption is made, that, whenever a disable request is issued, no longer other requests can be issued (including other disable requests). However, it must still be possible to receive confirmations of previously sent requests since they denote the completion of a service-element. The reception of indications however, shall not be allowed anymore, since they require to be dealt with by the Managed Object. The reception of either a disable confirmation or a disable indication, denotes the permanent end of the association in question, and is dealt with in the aborting state.
Process `initiaterelease` monitors the communication of a disable request, and if one occurs, communication is further restricted by process `releasing`.

```
  PROCESS initiaterelease [OSAP] (x : AI) : NOEXIT :=

    OSAP !x ?p : OSP ?aa : AID ;
    ([not(IsDisableReq(p))] -> initiaterelease[OSAP](x)
    []
     [IsDisableReq(p)] -> releasing[OSAP](x)
    )
  WHERE

    PROCESS releasing [Osap] (y : AI) : NOEXIT :=

      OSAP !y ?p : OSP ?aa : AID [IsConf(p) and not(IsDisable(p))];
      releasing [Osap] (y)

    ENDPROC (* releasing *)

  ENDPROC (* initiaterelease *)
```

The Aborting State

Termination of an association can be initiated both by the Manager and by the Agent. In this phase, the Managed Object has received either a disable indication or a disable confirmation, and is therefore aborting the association. This is modelled in the processes `manager` and `agent`, using the disable operator `[>`. The process that deals with the abortion is the process `forcerelease`, which is defined as follows :

```
PROCESS forcerelease[OSAP](x : AI) : EXIT :=

  OSAP !x !disableconf ?aa : AID ;
  EXIT
[]
  OSAP !x !disableind ?aa : AID ;
  EXIT

ENDPROC (* forcerelease *)
```

The possibility that both Manager and Agent or Service Provider terminate the association at the same time, is still covered by allowing only a disable indication to denote the end of the association. The corresponding confirmation will, in that case either be of importance for the Service Provider, or will never arive. Note that the abortion always succeeds!

The final local constraint is expressed by process `syncdisable`. This process monitors the occurance of a disable request and sees to it that the possible orderings between the disable service primitives is warranted. This means that it takes care that no disable confirmation occurs without having issued a disable request prior.

```
PROCESS syncdisable [OSAP] (x : AI) : EXIT :=

  OSAP !x ?p : OSP ?aa : AID [not(Isdisable(p) or Isenable(p))];
  syncdisable [OSAP] (x)
[]
  OSAP !x !disableind ?aa : AID;
  EXIT
[]
  OSAP !x !disablereq ?aa : AID;
  syncreleasing [OSAP] (x)

WHERE

  PROCESS syncreleasing [OSAP] (x : AI) : EXIT :=

    OSAP !x !disableind ?aa : AID;
    EXIT
  []
    OSAP !x !disableconf ?aa : AID;
    EXIT
  []
    OSAP !x ?p : OSP ?aa : AID [not(Isdisable(p) or Isenable(p))];
    syncreleasing [OSAP] (x)

  ENDPROC (* syncreleasing *)

ENDPROC (* syncdisable *)
```

This concludes the local constraints for the OM-service model. Note that several simplifications have been applied to accomodate for the restrictions the toolbox Eucalyptus poses on the Lotos-language. Both the number of messages in the communicating state and the number of concurrent associations have been severely limited.

The End-To-End Constraints

As mentioned before, the above processes specify the local ends of an association (one for the Agent and the other for the Manager). The actual synchronisation (end-to-end specification) is expressed by the following process:

```
PROCESS sync[OSAP](aa : aid) : NOEXIT :=

  pcsync[OSAP](aa)
|||
  ucsync[OSAP](aa)
|||
  unsync[OSAP](aa)
|||
  mnssync[OSAP](aa)

WHERE
```

Again, this process is divided into several independent processes, each representing the end-to-end constraints on different sets of service-elements. The distinction that is made, is done according to whether a service-element is user confirmed, provider confirmed, or unconfirmed. The process `pcsync` synchronizes the establishing and relinquishing of an association.

Synchronisation of User Confirmed Service Primitives

The process describing the synchronisation between the user confirmed service primitives is the process `ucsync`. Only one user confirmed service element is considered: the get service element, as can clearly be observed in the process description.

```
PROCESS ucsync [OSAP] (aa : aid) : EXIT :=

  (
  OSAP !initiator(aa) !getreq !aa ;
  OSAP !responder(aa) !getind !aa ;
  STOP
|||
  OSAP !responder(aa) !getres !aa ;
  OSAP !initiator(aa) !getconf !aa ;
  STOP
  )
[>
  ucsync [OSAP] (aa)

ENDPROC (* ucsync *)
```

Synchronisation of Unconfirmed Service Primitives

The end-to-end constraints on the order of the unconfirmed service primitives is equaly straightforward as the end-to-end constraints for the user confirmed service primitives, and is described by the following process:

```
PROCESS mnssync [OSAP] (aa : aid) : NOEXIT :=

 (OSAP !responder(aa) !reportreq !aa ;
  OSAP !initiator(aa) !reportind !aa ;
```

```
      STOP)
    [>
      mnssync [OSAP] (aa)

    ENDPROC (* mnssync *)

    PROCESS ucsync [OSAP] (aa : aid) : NOEXIT :=

     (OSAP !initiator(aa) !usetreq !aa ;
      OSAP !responder(aa) !usetind !aa ;
      STOP)
    [>
      ucsync [OSAP] (aa)

    ENDPROC (* ucsync *)
```

Synchronisation of Provider Confirmed Service Primitives

The process `pcsync` describes both the initiation of an association, and, the relinquishing of the association, which always succeeds. This abortion of the association has to take into account that there is the possibility that the service provider initiates a disable. Since only one disable indication to a Managed Object is allowed, the process that describes the possible ways to terminate an association is quite complicated.

```
    PROCESS pcsync [OSAP] (aa : aid) : NOEXIT :=

      sinitiate[OSAP](aa,initiator(aa), responder(aa))
    |||
      srelease [OSAP] (aa)

    WHERE
```

For simplicity's sake, process `pcsync` is divided into several subprocesses, taking care of only parts of the total functionality the process `pcsync` offers. The initialization is described in process `sinitiate`. This process is described below:

```
      PROCESS sinitiate[OSAP](a : aid, x,y : AI) : EXIT :=

        OSAP !x !enablereq !a ;
        ( OSAP !x !enableconf !a !false;
          sinitiate [OSAP] (a,x,y)
        []
          OSAP !y !enableind !a ;
          OSAP !x !enableconf !a !true;
          sinitiate [OSAP] (a,x,y)
        )

      ENDPROC (* sinitiate *)
```

The process dealing with the relinquishing of the association, i.e. process `srelease`, is described below. This process describes the disabling of both the Manager and the Agent. Since the abortion is symmetrical, this can be written using the interleaving operator `|||` and a process taking care of the abortion of only one component.

```
      PROCESS srelease [OSAP] (a : aid) : NOEXIT :=

        release [OSAP] (a, initiator(a), responder(a) )
      |||
        release [OSAP] (a, responder(a), initiator(a) )

      WHERE

        PROCESS release [OSAP] (a : aid, x,y : AI) : NOEXIT :=

         ((OSAP !x !disablereq !a;
           OSAP !y !disableind !a;
           (i;EXIT [] OSAP !x !disableconf !a; EXIT)
          )
        [> OSAP !x !disableind !a; EXIT )
        >>
          release [OSAP] (a, x , y)

        ENDPROC (* release *)

      ENDPROC (* srelease *)

    ENDPROC (* pcsync *)

  ENDPROC (* sync *)

ENDSPEC
```

Process `release` is very complicated because of the provider initiated disable that is allowed. This interferes with the disable indication that is the result of a previously communicated disable request.

2.4 Summary

In this Chapter the OM-service model was presented. The information found in Sections 2.1 and 2.2 was mainly based on [AVV97], and [Kap91]. Section 2.3 was based on an OM-service specification in LOTOS, made first, which can be found in Appendix A. This specification was based on requirements found in [AVV97], knowledge within CMG and some assumptions based on common sense. Several simplifications were made with respect to this specification, notably concerning:

- The number of concurrent associations, which reduced to one single association.
- The number of messages in the communicating state that can be in transit is vastly reduced.
- The number of service primitives is reduced.

One reason for this is the reduction of the state space which is thus created. Another reason for these simplifications is the limitations of the toolbox EUCALYPTUS, which does not support the full LOTOS constructs. The language accepted by the tool CÆSAR is only a subset of full LOTOS: no process recursion is allowed on the left and right hand part of the parallel-operator `|[...]|`, nor on the left hand part of the enable-operator `>>` and the disable-operator `[>`.

The distinction between the two types of service, the MOS and the MNS, is felt to be somewhat artificial: the impact both MOS and MNS have on the OM-service provider is the same, so to the OM-service provider treats service elements from the MOS and the MNS the same.

Chapter 3

The RR-service Model

This Chapter will discuss the RR-service model. In Section 3.1 the service elements will be determined and Section 3.2 will describe the dynamic behaviour of the RR-service model using LOTOS.

3.1 Determining the Service Elements and Service Primitives for the RR-service

The RR-service is used by an OM-protocol entity to send and receive OM Protocol Data Units (PDUs) to another OM-protocol entity. Figure 2.3 on page 8 clearly shows how the RR-protocol entity is situated with respect to the OM-protocol entity and the presentation layer. The RR-service enables the service user (i.e. an OM-protocol entity) to issue a request for an invocation of a predefined operation by a peer service user, and receive the response of that operation. Furthermore, the RR protocol layer supports the exchange of event reports and notification reports. To this purpose, six service elements are defined in [AVV97], being the following :

invoke, result, error, reject, addressed_eventreport, non-addressed_eventreport, notification.

Again, in [AVV97] a distinction is made between, in this case, addressed eventreports and non-addressed eventreports. The notification service element again is only non-addressed. Subscriptions are used to determine which protocol entities are to receive these non-addressed eventreports and notifications. Although now described at a lower level, these service elements still cause the same problems as before in chapter 2. Due to the underspecification in [AVV97], in this thesis the non-addressed eventreport service element shall be omitted and the notification service element is considered to be addressed only.

In [AVV97], again no distinction is made between the notion of service-elements and service-primitives, and the kind of service that is provided is not mentioned. It is assumed that an unconfirmed service is used. Since the RR-service is situated on top of a TCP/IP network, it is assumed, the RR-service offers reliable data transmission. According to [AVV97], a connectionless service is offered.

The terminology used in [AVV97] is as follows: an *invoker* is a protocol entity that initiated an invoke request carrying a specific operation and a *performer* is the protocol entity that is to perform an operation carried by an invoke service element.

The meaning associated to the service elements is, according to [AVV97] as follows:

- The invoke service element is used by the invoker to send an invoke request to a specific

Service Element	Service Primitive	type of service
invoke	invoke.req invoke.ind	unconfirmed
result	result.req result.ind	unconfirmed
error	error.req error.ind	unconfirmed
reject	reject.req reject.ind	unconfirmed
eventreport	eventreport.req eventreport.ind	unconfirmed
notification	notification.req notification.ind	unconfirmed

Table 3.1: *The service elements and service primitives of the RR-service*

performer.

- The result service element is used by a performer to send the results of a *previously* requested and successfully performed operation to the invoker.
- The error service element is used by the performer to send the results of a previously requested and unsuccessfully performed operation to the invoker.
- The reject service element is used by the performer to notify the invoker that the previously requested operation could not be performed.
- The eventreport service element enables a service user to send event reports to a specified destination.
- The notification service element enables a service user to notify the reception of an event report

A subtle distinction between the error service element and the reject service element exists: the error service element is only used when a performer tries to perform an operation, but does not succeed in performing this operation successfully. The reject service element is used when a performer does not even begin performing an operation.

Table 3.1 shows the relationship between the service primitives and the corresponding service element, and the kind of service that is offered. Again, note that this table does not specify (or imply) any dynamic behaviour.

In the RR-service model, not every service element described in table 3.1 is modelled. This is done in order to obtain a model that is as compact as possible, thereby easing the validation and verification that is to be performed on the model. The notification service element is omitted in the RR-service model because the (OM) notification service element was already omitted in the OM-service model, and as such, no need for the (RR) notification service element exists. A second alteration that is made is the omission of the error service element. This is done because it is felt that the distinction between the result service element and the error service element, as specified in [AVV97], is made at a level too low: the OM-protocol entities are not performing operations, instead the OM-service users are performing operations. Thus, to the OM-protocol entities, it is of no importance whether the results of an operation are communicated utilizing error service elements or result service elements.
The resulting service elements that are considered in the RR-service model thus are the following:

Service Element	Service Primitive	type of service
invoke	invoke.req invoke.ind	unconfirmed
result	result.req result.ind	unconfirmed
reject	reject.req reject.ind	unconfirmed
eventreport	eventreport.req eventreport.ind	unconfirmed

Table 3.2: *The service elements and service primitives of the RR-service model*

invoke, result, reject, eventreport.

As a consequence, the number of service primitives is reduced as well. The relation between the service primitives and the corresponding service element, is depicted in table 3.2.

3.2 The Dynamic Specification of the RR-service Model

This Section shall describe the dynamic behaviour of the RR-service model, and shall include the Abstract Data Types. The specification is written using the constraint oriented style.

Again, a single gate, `RSAP` (short for *RR Service Access Point*) is used to represent the complete RR-service boundary. The event structure at gate `RSAP` can be described as follows:

```
  RSAP <var: AI> <var: RSP> <var: AI> <var: AI> <var: OMPDU> <var: AId> <var: Nat>
```

The first parameter is used to identify the address of a service user, the second parameter is used to identify the service primitive that is communicated over gate `RSAP`. The third and fourth parameters are used to respectively identify the addresses of the invoker and the performer, whereas the fifth parameter is used to identify the OM Protocol Data Unit that is to be transported from the invoker to the performer or vice versa. The sixth parameter identifies the association this OM PDU uses, and the last parameter is used by the OM-protocol entity to make a distinction between different sessions.

The reason the event structure for gate `RSAP` is this big and unstructured, is due to the problems encountered with CÆSAR, mentioned in Chapter 2 on page 16.

The specification of the RR-service model is that of a never terminating one, written as follows:

```
SPECIFICATION RR_Service[RSAP] : NOEXIT
```

This specification uses the standard libraries `boolean` and `naturalnumber`.

```
LIBRARY
  boolean,
  naturalnumber
ENDLIB
```

The Abstract Data Types, used in the descriptions in Section 3.2.2, will be specified in the subsequent Section.

3.2.1 The Abstract Data Types used in the RR-service Model

The RR-service model requires the specification of several ADTs. However, some of these ADTs are already specified in Section 2.3.1, and for this reason, have been omitted in this Section. The ADTs that are omitted are the type `Address_Identifier`, specifying the sort `AI`, the type `Association_Identifier` and the type `SixteenTuplet`.
The first ADT that is introduced in this Section, is the ADT `RR_Service_Primitives`, specifying the sort `RSP`. This sort represents the RR-service model service primitives determined in Section 3.1. Like sort `OSP`, defined in Section 2.3.1, this sort was significantly simplified according to the needs of CÆSAR.ADT.

```
TYPE RR_Service_Primitives

IS
SORTS RSP
OPNS invokereq          (*! constructor *)   : -> RSP
     invokeind          (*! constructor *)   : -> RSP
     resultreq          (*! constructor *)   : -> RSP
     resultind          (*! constructor *)   : -> RSP
     rejectreq          (*! constructor *)   : -> RSP
     rejectind          (*! constructor *)   : -> RSP
     eventreportreq     (*! constructor *)   : -> RSP
     eventreportind     (*! constructor *)   : -> RSP

ENDTYPE (* RR_Service_Primitives *)
```

Again, the ADT `SixteenTuplet` is used to enumerate the abovedefined sort. Furthermore, some functions are introduced to identify service primitives as follows:

```
TYPE RSP_Classifier

IS SixteenTuplet, RR_Service_Primitives
OPNS Map               : RSP -> Tuplet
     IsInvokeReq       : RSP -> Bool
     IsInvokeInd       : RSP -> Bool
     IsResultReq       : RSP -> Bool
     IsResultInd       : RSP -> Bool
     IsRejectReq       : RSP -> Bool
     IsRejectInd       : RSP -> Bool
     IsEventreportreq  : RSP -> Bool
     IsEventreportind  : RSP -> Bool
EQNS
FORALL prim : RSP
OFSORT Tuplet
       Map(InvokeReq)          = One;
       Map(InvokeInd)          = Two;
       Map(ResultReq)          = Three;
       Map(ResultInd)          = Four;
       Map(RejectReq)          = Five;
       Map(RejectInd)          = Six;
       Map(EventreportReq)     = Seven;
       Map(EventreportInd)     = Eight;
OFSORT Bool
       IsInvokeReq(prim)       = Map(prim) eq One;
       IsInvokeInd(prim)       = Map(prim) eq Two;
       IsResultReq(prim)       = Map(prim) eq Three;
       IsResultInd(prim)       = Map(prim) eq Four;
       IsRejectReq(prim)       = Map(prim) eq Five;
       IsRejectInd(prim)       = Map(prim) eq Six;
       IsEventreportReq(prim)  = Map(prim) eq Seven;
       IsEventreportInd(prim)  = Map(prim) eq Eight;

ENDTYPE (* RSP_Classifier *)
```

A distinction towards different classes of `RSP`

Again, a classification can be made in the sort `RSP`. The first classification is the distinction between *requests* and *indications*. The ADT, specified below introduces two functions that make this distinction.

```
TYPE RSP_Servicetype

IS RSP_Classifier
OPNS IsReq             : RSP -> Bool
     IsInd             : RSP -> Bool
EQNS
FORALL prim : RSP
OFSORT Bool
       IsReq(prim)     = IsInvokeReq(prim) or IsResultReq(prim) or
                         IsRejectReq(prim) or IsEventReportReq(prim);
       IsInd(prim)     = IsInvokeInd(prim) or IsResultInd(prim) or
                         IsRejectInd(prim) or IsEventReportInd(prim);

ENDTYPE (* RSP_ServiceType *)
```

The other classification that can be made is a classification towards the service elements. The ADT `RSP_Elements` is used to express this classification.

```
TYPE RSP_Elements

IS RSP_Classifier
OPNS IsInvoke          : RSP -> Bool
     IsResult          : RSP -> Bool
     IsReject          : RSP -> Bool
     IsEventreport     : RSP -> Bool
EQNS
FORALL prim : RSP
OFSORT Bool
       IsInvoke(prim)       = IsInvokeReq(prim) or IsInvokeInd(prim);
       IsResult(prim)       = IsResultReq(prim) or IsResultInd(prim);
       IsReject(prim)       = IsRejectReq(prim) or IsRejectInd(prim);
       IsEventreport(prim)  = IsEventreportReq(prim) or IsEventreportInd(prim);

ENDTYPE (* RSP_Elements *)
```

Pairing of service primitives

The subsequent ADT shall be used in the dynamic descriptions in Section 3.2.2 to match two service primitives. The ADT `RSP_Match` defines relations between certain service primitive requests and service primitive indications as follows:

```
TYPE RSP_Match

IS RSP_Classifier
OPNS _IndForReq_        : RSP, RSP -> Bool
EQNS
FORALL p,q : RSP
OFSORT Bool
```

```
       q IndForReq p   = (IsInvokeReq(p) and IsInvokeInd(q)) or
                         (IsResultReq(p) and IsResultInd(q)) or
                         (IsRejectReq(p) and IsRejectInd(q)) or
                         (IsEventreportReq(p) and IsEventReportInd(q));

ENDTYPE (* RSP_Match *)
```

The abovedefined types deal with the representation, identification and classification of RR service primitives.
Next, the OM PDUs need to be described. To this end, the ADT **Dataunits** is specified. This type introduces a sort **OMPDU** which represents a set of OM Protocol Data Units (see Chapter 4).

```
TYPE Dataunits

IS
SORTS OMPDU
OPNS Uset      (*! constructor *)   : -> OMPDU
     Enable    (*! constructor *)   : -> OMPDU
     Disable   (*! constructor *)   : -> OMPDU
     Report    (*! constructor *)   : -> OMPDU
     Get       (*! constructor *)   : -> OMPDU

ENDTYPE (* Dataunits *)
```

The ADT **SixteenTuplet** is again used to enumerate the abovedefined sort:

```
TYPE OMPDU_Classification

IS Dataunits, SixteenTuplet
OPNS Map                : OMPDU -> Tuplet
EQNS
OFSORT Tuplet
       Map(Uset)        = One;
       Map(Enable)      = Two;
       Map(Disable)     = Three;
       Map(Report)      = Four;
       Map(Get)         = Five;

ENDTYPE (* OMPDU_Classification *)
```

This concludes the Abstract Data Types definitions for the RR-service model specification.

3.2.2 The Dynamic Behaviour of the RR-service Model

The RR-service, as described in [AVV97], is a connectionless service which offers reliable data communications to take place, and it depends on the underlying layers to do so. A connectionless service dictates that two subsequent messages sent from A to B can involve message reordering.

Thus the requirements the RR-service model is based upon are the following:

1. A reliable service is offered.
2. At most four messages can be in transit at any point in time.

As mentioned in Section 3.1, a distinction is made between invokers and performers. Severe simplifications were made: instead of offering a possibly unbounded number of concurrent requests for invokes, responses, rejects and eventreports, the number of messages in transit is severely reduced. This is due to the limitations of CÆSAR. However, with respect to reality, this can be commented on by pointing out that up till now, no medium has been created that can process an infinite number of messages concurrently.

These requirements are expressed by the following top level process:

```
BEHAVIOUR

  communications [RSAP]

WHERE
```

This top level process, process **`communications`**, specifies exactly four communications, which is expressed as follows:

```
  PROCESS communications [RSAP] : NOEXIT :=

    communication [RSAP] ||| communication [RSAP] |||
    communication [RSAP] ||| communication [RSAP]


  WHERE
```

The decomposition of one communication

The process **`communication`** describes the local end requirements and the end-to-end constraints of one communication. The local end constraints (which are very elementary) are expressed by the processes **`invoker`** and **`performer`**, while the end-to-end constraints are expressed by the process **`syncip`**. The process **`communication`** is described as follows:

```
    PROCESS communication [RSAP] : NOEXIT :=

      (invoker [RSAP] ||| performer [RSAP])
    ||
      syncip [RSAP]

    WHERE
```

The Local End Constraints

As mentioned before, the local end constraints for a communication are very elementary, as the following specification clearly shows:

```
      PROCESS invoker [RSAP] : NOEXIT :=

        RSAP ?n : AI ?s : RSP ?k : AI ?l : AI ?p : OMPDU ?a : AId ?j : nat
             [(IsResultInd(s) or IsRejectInd(s)) and (n eq l)];
        invoker [RSAP]
      []
        RSAP ?n : AI ?s : RSP ?k : AI ?l : AI ?p : OMPDU ?a : AId ?j : nat
             [(IsEventreportReq(s) or IsInvokeReq(s)) and (n eq k)];
        invoker [RSAP]

      ENDPROC (* invoker *)
```

The process **`invoker`** specifies the communication of either a result indication or a reject indication, or the communication of either an eventreport request or an invoke request. The process **`performer`** specifies the counterpart requests and indications and is equally elementary:

```
      PROCESS performer [RSAP] : NOEXIT :=

        RSAP ?n : AI ?s : RSP ?k : AI ?l : AI ?p : OMPDU ?a : AId ?j : nat
             [(IsResultReq(s) or IsRejectReq(s)) and (n eq l)];
        performer [RSAP]
      []
        RSAP ?n : AI ?s : RSP ?k : AI ?l : AI ?p : OMPDU ?a : AId ?j : nat
             [(IsEventreportInd(s) or IsInvokeInd(s)) and (n eq k)];
        performer [RSAP]

      ENDPROC (* performer *)
```

The End-To-End Constraint

Sometimes the end-to-end constraints can be hard to specify. However, with a service this elementary, no difficulties shall be encountered. The process `syncip` requires no explanation and is given as is:

```
      PROCESS syncip [RSAP] : NOEXIT :=

        RSAP ?n : AI ?s : RSP ?k : AI ?l : AI ?p : OMPDU ?a : AId ?j : nat
             [IsReq(s) and not (k eq l)];
        RSAP !l ?t : RSP !k !l !p !a !j
             [t IndForReq s];
        syncip [RSAP]

      ENDPROC (* syncip *)

    ENDPROC (* communication *)

  ENDPROC (* communications *)

ENDSPEC
```

3.3 Summary

In the previous Sections, the RR-service model is presented. Note that a constraint oriented specification is given in Section 3.2.2 but the RR-service model could equally well have been described using the monolythic specification style.

Section 3.1 is mainly based on [AVV97]. Some adaptions were made in the service specification (e.g. the non-addressed service elements were omitted), since [AVV97] was too obscure concerning the specification of these service elements. The RR-service model is a further adaption to the service specification, found in Appendix B, on the following points:

- The error and notification service elements have been omitted.
- The number of messages in transit is reduced to four.

The latter item is again due to CÆSAR. Note that it is easy to extend the RR-service model, such that it describes more than four (or less than four) communications by removing or adding an extra instance of process `communication` interleaved with the other `communication` processes.

Chapter 4

The OM-protocol Entity Model

This Chapter describes the OM-protocol entity model. The first Section, Section 4.1 describes the environment of the OM-protocol entity model. Section 4.2 describes the decomposition techniques used to describe the OM-protocol entity model, and Section 4.3 then describes the dynamic behaviour of the OM-protocol entity model.

4.1 The OM-protocol Entity Model Environment

The OM-protocol entity is a protocol entity whose main purpose is to receive service primitives from the service user, and translate them into service primitives the RR service layer can handle. Thus it fills in the gap between the OM-service and the RR-service, and is used to add functionality to the RR-service and mask unwanted properties of the RR-service. The main task of the OM-protocol entity is to make sure no unexpected behaviour is offered at both service boundaries.
The OM-protocol entity is bordered on two sides, as depicted in figure 4.1. One of the borders is the service user border, represented by the gate `OSAP`, and the other is the underlying service provider border, represented by the gate `RSAP`.

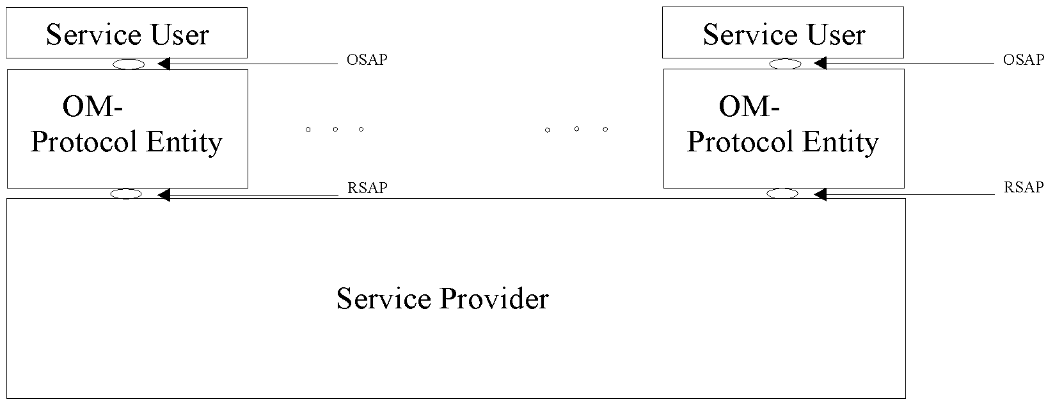

Figure 4.1: *The OM-protocol entity and its borders*

4.2 Decomposition Techniques

Writing a specification for a protocol entity without using some form of decomposition technique almost always proofs to be too great a challenge. Several techniques exist to deal with writing such extensive specifications. The decomposition technique used in this Chapter is discussed in this Section. The Abstract Data Types used in the descriptions for the dynamic behaviour of the OM-protocol entity model are described in Section 4.3.1 and the dynamic behaviour of the OM-protocol entity model is described in Section 4.3.2.

The service user can communicate with the OM-protocol entity by means of service primitives. These service primitives have to be translated into service primitives the RR-layer can use. The protocol entity shall be specified using the techniques described in [SPKV93], which prescribes a step-by-step design. The first step in this technique, is to distinguish two separate functional units, which, by *separation of concerns* can be sufficiently modelled in a constraint oriented way. The protocol entity is therefore split into two functional units: one for communication with the service user, and one for the communication with the service provider. The first one contains the so-called Upper Protocol Functions and is therefore named the UPF and the second functional unit contains the Lower Protocol Functions, and is appropriately called the LPF. This decomposition is depicted in figure 4.2.

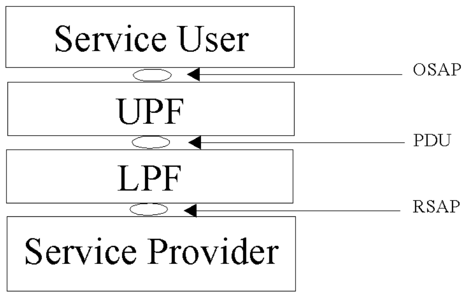

Figure 4.2: *The decomposition of the protocol entity into the UPF and LPF*

The UPF and the LPF interact on a shared gate, namely `PDU`. This gate serves as a gateway between the RR and the OM service boundaries, and synchronisation of the two layers occurs here. The main functionality of the UPF is enforcing sequences of service primitives on the service users, whereas the LPF describes the functionalities the protocol entity must provide, such as defragmentation, multiplexing, and error-free transmission using lower layers functionality, but also the handling of provider initiated termination.

The UPF and the LPF are decomposed again in two separate processes, one for the internal interaction on the `PDU` gate, and one for the synchronisation with respectively the service user and the service provider. Note that the introduced gate `PDU` serves as a global gate in this case, but could very well be split into several independent gates, all fulfilling the role of the `PDU` gate. These independent gates can be used to make a clear distinction between different events and communications.
More information concerning the techniques sketched in this Section can be found in [SPKV93].

4.3 The Specification of the OM-protocol Entity Model

The OM-protocol entity model is described using a mixture of a constraint oriented and resource oriented specification style. The service boundaries for the OM-protocol entity model are represented by gates `OSAP` and `RSAP`. The event structures are described as follows:

```
OSAP <var: AI> <var: OSP> <var: AId>
OSAP <var: AI> <var: OSP> <var: AId> <var: BOOL>
```

```
  RSAP <var: AI> <var: RSP> <var: AI> <var: AI> <var: OMPDU> <var: AId> <var: Nat>
```

The gates are parameterized as described in Sections 2.3 and 3.2. The specification of the OM-protocol entity model is that of a never terminating one, written as follows:

```
SPECIFICATION OM_Protocol_Entity [OSAP, RSAP] : NOEXIT
```

This specification again uses the standard libraries `boolean` and `naturalnumber`.

```
LIBRARY
  boolean,
  naturalnumber
ENDLIB
```

The subsequent Section will specify the Abstract Data Types, used in the descriptions in Section 4.3.2.

4.3.1 The Abstract Data Types used in the OM-protocol Entity Model

This Section consists of the ADTs that are needed in the LOTOS descriptions in Section 4.3.2. However, most ADTs have already been specified in Sections 2.3.1 and 3.2.1, and hence shall be omitted in this Section. The only additional ADTs that are needed are the ADT `PDU_Kind` and `PDU_Identifier`. The ADT `PDU_Kind` defines a sort with four elements. These elements are used in the OM-protocol entity model to identify the "direction of travel" of an OM PDU (e.g. from the `OSAP` boundary to the `RSAP` boundary and vice versa).

```
TYPE PDU_Kind

IS
SORTS Kind
OPNS req     (*! constructor *) : -> Kind
     ind     (*! constructor *) : -> Kind
     res     (*! constructor *) : -> Kind
     conf    (*! constructor *) : -> Kind

ENDTYPE (* PDU_Kind *)
```

No equivalence is specified on the elements of this sort, because the elements of this sort are only used for value passing over gates. For this, LOTOS uses ADT equivalence.

The ADT `PDU_Identifier` is used to identify the PDUs, being an enable, disable, etc.

```
TYPE PDU_Identifier

IS OMPDU_Classification
OPNS IsEnable        : OMPDU -> Bool
     IsDisable       : OMPDU -> Bool
     IsGet           : OMPDU -> Bool
     IsUset          : OMPDU -> Bool
     IsReport        : OMPDU -> Bool
EQNS
FORALL pdu : OMPDU
OFSORT Bool
       IsUset(pdu)      = Map(pdu) eq One;
       IsEnable(pdu)    = Map(pdu) eq Two;
       IsDisable(pdu)   = Map(pdu) eq Three;
       IsReport(pdu)    = Map(pdu) eq Four;
       IsGet(pdu)       = Map(pdu) eq Five;

ENDTYPE (* PDU_Identifier *)
```

This concludes the ADTs used in the OM-protocol entity LOTOS-description.

4.3.2 The Dynamic Behaviour of the OM-protocol Entity Model

The OM-protocol entity is based on a few requirements. None of these requirements have been found in [AVV97], since no requirements for the OM-protocol entity were stated in [AVV97]. Therefore, all requirements have been assumed, yet these assumptions are based on one or more arguments. The following list states the requirements, and the arguments for assuming this is required.

1. It is assumed the OM-protocol entity accesses the RR-service using only one Service Access Point. Since the RR-service offers only a mere datagram service, it is expected the RR-service will not delay progress of the OM-protocol entities.
2. The addressing mechanism for the OM-protocol entity and the RR-service is unclear. It is assumed the OM-protocol entity address at the gate `OSAP` is identical to the address at the gate `RSAP`. The reason for this assumption is simplicity. A more elaborate mapping than the identical mapping can be introduced at the cost of readability.
3. The OM-protocol entity does not fragment and defragment messages received from the service users and the service provider. This functionality is not felt to be part of the functionality the application layer in the OSI-model is supposed to offer.
4. The OM-protocol entity does not provide a retransmission mechanism. No need for this mechanism is needed, since this is already taken care of in the TCP/IP layer. Furthermore, the RR-service offered assumes reliable communications.

The OM-protocol entity is modelled after the simplifications applied to the OM-service. This means that one OM-protocol entity maintains only one association. This, together with the abovementioned requirements are expressed in the following top-level process:

```
BEHAVIOUR

  association [OSAP, RSAP] (a,n)

WHERE
```

Note that the variables `a` and `n` have to be instantiated in this model with values of type `AId` and `AI`. The first variable (variable `a`) represents the association and the second variable (variable `n`) represents the (unique) address of the protocol entity.
A more general OM-protocol entity, that is able to maintain more than one association at a time is easily expressed by the following top-level process:

```
BEHAVIOUR

  associations [OSAP, RSAP] (A,n)

WHERE

  PROCESS associations [OSAP, RSAP] (A : set_of_AId, n : AI) : NOEXIT :=

    CHOICE aa : AId [] [aa isin A] ->
    (association [OSAP, RSAP] (aa,n)
  |||
     i; associations [OSAP, RSAP] (remove(aa,A), n) )

  ENDPROC (* associations *)
```

This kind of description, however, cannot be dealt with by CÆSAR, and as such, again only one association is considered.

The decomposition of one association

Analogous to Section 2.3.2, the decomposition of one association is considered in isolation. The decomposition techniques discussed in Section 4.2 are used. Furthermore, the OM-protocol entity makes a distinction to whether the side of the association that is considered is in a role of Manager, or in the role of the Agent.

```
PROCESS association [OSAP, RSAP] (aa : AId, n : AI) : NOEXIT :=

  HIDE INIT, CON, UNC, MNS, REL, ABORT IN (
  [n eq initiator(aa)] ->
  (
    (mupfs [OSAP, INIT, CON, UNC, MNS, REL, ABORT] (aa,n)
  |[OSAP]|
     manager [OSAP] (aa)
    )
  |[INIT, CON, UNC, MNS, REL, ABORT]|
    (mlpfs [RSAP, INIT, CON, UNC, MNS, REL, ABORT] (aa,n,0)
  |[RSAP]|
    rrsi [RSAP] (n,aa)
    )
  )
[]
  [n eq responder(aa)] ->
  (
    (aupfs [OSAP, INIT, CON, UNC, MNS, REL, ABORT] (aa,n)
  |[OSAP]|
     agent [OSAP] (aa)
    )
  |[INIT, CON, UNC, MNS, REL, ABORT]|
    (alpfs [RSAP, INIT, CON, UNC, MNS, REL, ABORT] (aa,n)
  |[RSAP]|
    rrsi [RSAP] (n, aa)
    )
  )
)

WHERE
```

Process **association** specifies the interaction between the Upper Protocol Functions (process **mupfs** for the Manager side of the association, and process **aupfs** for the Agent side of the association), and the Lower Protocol Functions (process **mlpfs** for the Manager side of the association, and process **alpfs** for the Agent side of the association). Processes **manager** and **agent** describe the local constraints at the **OSAP** gate for the OM-protocol entity, and are as specified in Section 2.3.2.
The process **rrsi** is used to describe the interface at the RR-service boundary. Since this behaviour is trivial, it is introduced without any comments:

```
PROCESS rrsi [RSAP] (n : AI, aa : AId) : NOEXIT :=

  RSAP !n ?s : RSP ?k : AI ?l : AI ?p : OMPDU !aa ?j : nat
       [not(k eq l) and (Iseventreport(s) eq Isreport(p)];
  rrsi [RSAP] (n,aa)

ENDPROC (* rrsi *)
```

The Upper Protocol Functions and the Lower Protocol Functions communicate over dedicated gates **INIT**, **CON**, **UNC**, **MNS**, **REL** and **ABORT**. Communication over these gates represents the passing of OM PDUs between the Upper Protocol Functions and the Lower Protocol Functions. First the Upper Protocol Functions shall be described. The Lower Protocol Functions will be described on page 46 and onward.

The Upper Protocol Functions

The process `mupfs` is structured much like the process `manager`. This is because the main communication takes place at the OM service boundary. The functionality this process exhibits is the translation of service primitives over gate `OSAP` into OM PDUs, and vise versa, the translation of OM PDUs into service primitives which again can be communicated over gate `OSAP`. The process `mupfs` is described as follows:

```
PROCESS mupfs [OSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa: AId, n: AI) : NOEXIT :=

  mu_initiate [OSAP,INIT] (aa,n)
>>
((mu_communicate [OSAP,CON,UNC,MNS] (aa,n) ||| u_release [OSAP,REL] (aa,n) )
[>
  u_abort [OSAP,ABORT] (aa,n) )
>>
  mupfs [OSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa,n)

ENDPROC (* mupfs *)
```

The process `aupfs` follows the same structure as the Manager counterpart process `mupfs` and is described as follows:

```
PROCESS aupfs [OSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa: AId, n: AI) : NOEXIT :=

  au_initiate [OSAP,INIT] (aa,n)
>>
((au_communicate [OSAP,CON,UNC,MNS] (aa,n) ||| u_release [OSAP,REL] (aa,n) )
[>
  u_abort [OSAP,ABORT] (aa,n) )
>>
  aupfs [OSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa,n)

ENDPROC (* aupfs *)
```

The processes used in the abovedescribed processes shall be further decomposed. Each process describes the translation of service primitives used at the gate `OSAP` into OM PDUs in different states (cf. figure 2.6).

The Initial State — Upper Part

In the initial state, a service user is able to request the establishment of an association, in which case he acts as a Manager, or a service user is offered an association, in which case he acts as an Agent. The service primitives that affect this state are the enable service primitives. These service primitives are passed on from the service boundary `OSAP` to the dedicated gate `INIT`, which represents the corresponding OM PDU, and vice versa. The event structure at this gate can be written as follows:

```
INIT <var: AId> <var: OMPDU>
INIT <var: AId> <var: OMPDU> <var: BOOL>
```

Note that again the polymorphic gate features of LOTOS are used. This is again due to the problem mentioned on page 16. The first parameter binds the OM PDU that is passed in this gate to the association that is used. The second parameter represents the OM PDU that is passed. The optional third parameter is used to denote the success or failure of the establishment of the association.

The process describing the upper protocol functions for the initial state for the Manager (process `mu_initiate`) is very straightforward:

```
    PROCESS  mu_initiate [OSAP,INIT] (aa: AID, n : AI) : EXIT :=

      OSAP !n !enablereq !aa;
      INIT !aa !enable;
      INIT !aa !enable ?b : Bool;
      OSAP !n !enableconf !aa !b;
      ( [b     ] -> EXIT
      []
        [not(b)] -> mu_initiate [OSAP,INIT] (aa,n)
      )

ENDPROC (* mu_initiate *)
```

In order to prevent the Manager from receiving an enable confirmation, prior to the Agent receiving the enable indication, an additional, internal communication using gate `INIT` is used to signal that the Agent has received the indication. No feed back by the Agent is requested, i.e. the agent does not have to give a response to the enable indication. A time-out mechanism is introduced to model the non-response of the Agent.

```
    PROCESS  au_initiate[OSAP,INIT](aa : aid, n : AI) : EXIT :=

      INIT !aa !enable;
      ( OSAP !n !enableind !aa;
        INIT !aa !enable !true;
        EXIT
      []
        i; (* TIME_OUT *)
        INIT !aa !enable !false;
        au_initiate[OSAP,INIT](aa,n)
      )

    ENDPROC (* au_initiate *)
```

Note that the only service element that is allowed to be passed on in the `INIT` gate is the enable service element.

These two processes define the Upper Protocol Functions in the initial state, for both the Manager and the Agent. Note that these two processes together represent the initial state of the protocol entity, since the entity is supposed to be able to assume both the role of a Manager as well as the role of the Agent.

The Communicating State — Upper Part

In the communicating state, messages can be sent from the Manager to the Agent and from the Agent to the Manager. Every message can either be classified as *user confirmed* or *unconfirmed*, and every unconfirmed message can itself be part of the MOS or the MNS (see Section 2.2). For reasons of clarity, this distinction shall be maintained by using different gates for these three classes of messages: the `CON` gate will be used for the user confirmed OM PDUs, the `UNC` gate shall be used for the unconfirmed MOS OM PDUs, and the MNS gate shall be used for the unconfirmed MNS OM PDUs. The event structure at the gates can be described as follows:

```
CON <var: AId> <var: OMPDU> <var: KIND>
UNC <var: AId> <var: OMDDU>
MNS <var: AId> <var: OMPDU>
```

Again, the first parameter for all three gates identifies the association that uses this gate, the second parameter represents the PDU passed in that gate, and the optional third parameter indicates the direction of travel in the protocol entity itself, by means of the sort KIND.

Two processes, mu_communicate and au_communicate describe the Upper Protocol Functions that are offered in this state for respectively the Manager and the Agent. The simplifications, made here are the same as those made in the OM-service model: only one user confirmed message is considered, being the get service element and two unconfirmed messages, being the set service element and the report service element.

```
PROCESS mu_communicate [OSAP,CON,UNC,MNS] (aa: aid, n: AI) : NOEXIT :=

  mu_con [OSAP,CON] (aa, n)
|||
  mu_unc [OSAP,UNC] (aa, n)
|||
  mu_mns [OSAP,MNS] (aa, n)

WHERE

  PROCESS mu_con [OSAP,CON] (aa : AId, n : AI) : NOEXIT :=

    OSAP !n !getreq !aa;
    CON !aa !get !req;
    STOP
  |||
    CON !aa !get !conf;
    OSAP !n !getconf !aa;
    STOP

  ENDPROC (* mu_con *)

  PROCESS  mu_unc [OSAP,UNC] (aa: AId, n: AI) : NOEXIT :=

    OSAP !n !usetreq !aa;
    UNC !aa !uset;
    STOP

  ENDPROC (* mu_unc *)

  PROCESS  mu_mns [OSAP,MNS] (aa: AId, n: AI) : NOEXIT :=

    MNS !aa !report;
    OSAP !n !reportind !aa;
    STOP

  ENDPROC (* mu_mns *)

ENDPROC (* mu_communicate *)
```

In the preceding processes, each OM PDU is dealt with by defining auxiliary processes, each describing the interaction between the OM service boundary and the internal gate dedicated to a special OM PDU. This way, a clear separation of concerns is maintained.

The Agent part of the Upper Protocol Functions for the communicating state is almost equivalent to that for the Manager, since the same obvious distinctions, similar to those of the Manager, can be made.

```
PROCESS au_communicate [OSAP,CON,UNC,MNS] (aa: AId, n: AI) : NOEXIT :=

  au_con [OSAP,CON] (aa, n)
|||
  au_unc [OSAP,UNC] (aa, n)
|||
  au_mns [OSAP,MNS] (aa, n)

WHERE

  PROCESS au_con [OSAP,CON] (aa: AId, n: AI) : NOEXIT :=

    CON !aa !get !ind;
    OSAP !n !getind !aa;
    STOP
  |||
    OSAP !n !getres !aa;
    CON !aa !get !res;
    STOP

  ENDPROC (* au_con *)

  PROCESS au_unc [OSAP,UNC] (aa: AId, n: AI) : NOEXIT :=

    UNC !aa !uset;
    OSAP !n !usetind !aa;
    STOP

  ENDPROC (* au_unc *)

  PROCESS au_mns [OSAP,MNS] (aa: AId, n: AI) : NOEXIT :=

    OSAP !n !reportreq !aa;
    MNS !aa !report;
    STOP

  ENDPROC (* au_unc *)

ENDPROC (* au_communicate *)
```

The Releasing state — upper part

The releasing state is triggered by the OM service user's invocation of a disable request. Local constraints specified in processes `agent` and `manager` take care that the communication at gate `OSAP` is restricted when entering this state. The lower protocol functions have to take care, that the passing on of OMPDUs over gates `CON`, `UNC` and `MNS` is restricted such that the OM-service is offered and no deadlock state is introduced. A dedicated gate `REL` is introduced for passing on the OM PDU representing the disable request communicated at gate `OSAP`. The event structure at the gate `REL` is as follows :

```
REL <var: AId> <var: OMPDU>
```

Here, again the first parameter identifies the association to be relinquished and the second parameter represents the PDU to be communicated over gate **REL**.
Only one process, process **u_release** describes the Upper Protocol Functions that are offered in this state for both the Manager and the Agent as follows:

```
PROCESS u_release [OSAP,REL] (aa: aid, n: AI) : NOEXIT :=

  OSAP !n !disablereq !aa;
  REL !aa !disable;
  STOP

ENDPROC (* u_release *)
```

Obviously, the abovedefined process is very elementary.

The Aborting State — Upper Part

This state is triggered by both a disable indication and a disable confirmation and marks the end of one session for the association. A dedicated gate, gate **ABORT** is used to communicate the OM PDU that indicates the end of a session. The event structure at this gate is as follows:

```
ABORT <var: AId> <var: OMPDU> <var: KIND>
```

Here, again the first parameter identifies the association to be released, and the second parameter represents the OM PDU that is communicated over this gate. The third parameter is used to identify the OM PDU as an indication or as a confirmation.

Again, only one process is used to describe the Upper Protocol Functions that are offered in this state for both the Manager and the Agent. These functions are expressed by process **u_abort** and are rather straightforward:

```
PROCESS u_abort [OSAP,ABORT] (aa: AId, n: AI) : EXIT :=

  ABORT !aa !disable !ind;
  OSAP !n !disableind !aa;
  EXIT
[]
  ABORT !aa !disable !conf;
  OSAP !n !disableconf !aa;
  EXIT

ENDPROC (* u_abort *)
```

This concludes the specification of the Upper Protocol Funcions. Most of these processes are rather straightforward: no translation of service primitives into OM PDUs and vice versa is specified, and as such, the Upper Protocol Functions are trivial. The Lower Protocol Functions, described next, will prove to be more challenging.

The Lower Protocol Functions

The top-level processes, dealing with the Lower Protocol Functions, are the processes **mlpfs** and **alpfs**. Their structure is much like the structures of processes **mupfs** and **aupfs**.

```
PROCESS mlpfs [RSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa: AId, n: AI, j: nat) :NOEXIT :=

  ml_initiate [RSAP,INIT] (aa,n,j)
>>
```

```
((ml_communicate [RSAP,CON,UNC,MNS] (aa,n,j)
|[CON,UNC,MNS]|
  ml_release [RSAP,CON,UNC,MNS,REL] (aa,n,j) )
[>
  l_abort [RSAP,ABORT] (aa,n,j) )
>>
  mlpfs [RSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa,n,succ(j))

ENDPROC (* mlpfs *)

PROCESS alpfs [RSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa: AId, n: AI) :NOEXIT :=

  al_initiate [RSAP,INIT] (aa,n)
>> ACCEPT j : nat IN
((al_communicate [RSAP,CON,UNC,MNS] (aa,n,j)
|[CON,UNC,MNS]|
  al_release [RSAP,CON,UNC,MNS,REL] (aa,n,j) )
[>
  l_abort [RSAP,ABORT] (aa,n,j) )
>>
  alpfs [RSAP,INIT,CON,UNC,MNS,REL,ABORT] (aa,n)

ENDPROC (* alpfs *)
```

Note the small difference in these processes: the Manager side of an association determines a so-called session-identifier, whereas the Agent side of the association can only accept this session-identifier after the association is established. This session-identifier is used to distinguish between the different sessions of an association, but is of no concern to the OM-service user and is therefore modelled in the Lower Protocol Functions.

The Initial State — Lower Part

The Lower Protocol Functions in the initial state must ensure the OM-protocol entity does not block communications at the `RSAP` boundary. The main task for the OM-protocol entity is to deal with PDUs related to the establishment of the association. The Lower Protocol Functions for the Manager side of the association are different from those of the Agent side of the association. Informally, the following events can occur in this state for the Manager side of the association:

- When an enable PDU is offered at the `INIT` gate, an invoke request service primitive is communicated, carrying the PDU and the current session-identifier. Prior to this event, every invoke indication is rejected using a reject request service primitive, and every other indication is discarded.

- When a response indication, carrying the current session-identifier and an enable PDU is communicated at the RR service boundary, the setup is to be considered successful. An enable PDU is constructed using the `INIT` gate, carrying the success indication.

- When a reject indication, carrying the current session-identifier and an enable PDU is communicated at the RR service boundary, the setup is to be considered unsuccessful. An enable PDU is constructed using the `INIT` gate, carrying the failure indication.

- Every invoke indication carrying a session-identifier other than the current session-identifier, shall be rejected using a reject request service primitive.

- Every reject-,response- and eventreport indication carrying a session-identifier other than the current session-identifier, shall be discarded.

A more exact description of events is described by the Lotos- description of process `ml_initiate`:

```
PROCESS ml_initiate [RSAP,INIT] (aa: AId, n: AI, j: nat) : EXIT :=

  ml_idle [RSAP] (aa,n)
[>
 (INIT !aa !enable;
  RSAP !n !invokereq !initiator(aa) !responder(aa) !enable !aa !j;
  ( ml_filter [RSAP] (aa,n,j)
  [>
   (  RSAP !n ?s : RSP !responder(aa) !initiator(aa) !enable !aa !j
             [IsResultInd(s) or IsRejectInd(s)];
     ([IsResultInd(s)] -> INIT !aa !enable !true;
                          EXIT
     []
       [IsRejectInd(s)] -> INIT !aa !enable !false;
                          ml_initiate [RSAP,INIT] (aa,n,j)
     )
   )
  )
 )

WHERE

  PROCESS ml_idle [RSAP] (aa: AId, n: AI) : NOEXIT :=

    RSAP !n !invokeind ?k : AI ?l : AI ?p : OMPDU !aa ?j : nat;
    RSAP !n !rejectreq !l !k !p !aa !j;
    ml_idle [RSAP] (aa,n)
  []
    RSAP !n ?s : RSP ?k : AI ?l : AI ?p : OMPDU !aa ?j : nat
         [not(IsInvokeInd(s)) and not(IsReq(s))];
    ml_idle [RSAP] (aa,n)

  ENDPROC (* ml_idle *)

  PROCESS ml_filter [RSAP] (aa: AId, n: AI, j: nat) : NOEXIT :=

    RSAP !n !invokeind ?k : AI ?l : AI ?p : OMPDU !aa ?m : nat
         [not (m eq j)];
    RSAP !n !rejectreq !l !k !p !aa !m;
    ml_filter [RSAP] (aa,n,j)
  []
    RSAP !n ?s : RSP ?k : AI ?l : AI ?p : OMPDU !aa ?m : nat
         [not(IsInvokeInd(s)) and not(IsReq(s)) and not(m eq j)];
    ml_filter [RSAP] (aa,n,j)

  ENDPROC (* ml_filter *)

ENDPROC (* ml_initiate *)
```

The events that can occur in the Agent side of the association are somewhat different from those for the Manager side. Informally, they can be described as follows:

- Every invoke indication, not carrying an enable PDU is rejected using the reject request service primitive.
- Every reject-,result- or eventreport indication is discarded.
- An invoke indication carrying an enable PDU and a session-identifier, causes the OM-protocol entity to communicate this PDU using the `INIT` gate. When the association is

accepted, a success PDU is communicated using the `INIT` gate, which is sent to the peer entity using a result request service primitive. When the association is rejected, a failure PDU is communicated using the `INIT` gate, which is sent to the peer entity using a reject request service primitive.

```
PROCESS al_initiate [RSAP,INIT] (aa: AId, n: AI) : EXIT(nat) :=

  al_idle [RSAP] (aa,n)
[>
  RSAP !n !invokeind !initiator(aa) !responder(aa) !enable !aa ?j : nat;
  INIT !aa !enable;
  INIT !aa !enable ?b : bool;
  ( [not(b)] -> RSAP !n !rejectreq !responder(aa) !initiator(aa) !enable !aa !j;
                al_initiate [RSAP,INIT] (aa,n)
  []
    [b     ] -> RSAP !n !resultreq !responder(aa) !initiator(aa) !enable !aa !j;
                EXIT(j)
  )

WHERE

  PROCESS al_idle [RSAP] (aa: AId, n : AI) : NOEXIT :=

    RSAP !n !invokeind !initiator(aa) !responder(aa) ?p : OMPDU !aa ?j : nat
         [not(IsEnable(p))];
    RSAP !n !rejectreq !responder(aa) !initiator(aa) !p !aa !j;
    al_idle [RSAP] (aa,n)
  []
    RSAP !n ?s : RSP !initiator(aa) !responder(aa) ?p : OMPDU !aa ?j : nat
         [not(IsInvokeInd(s)) and not(IsReq(s))];
    al_idle [RSAP] (aa,n)

  ENDPROC (* al_idle *)

ENDPROC (* al_initiate *)
```

Note that, when the Agent did accept the association, process `al_initiate` exits with the current session-identifier, thus making this identifier available to processes representing the other states of the association. The additional processes (i.e. `ml_idle`, `ml_filter` and `al_idle`) are needed to avoid the RR-layer from cluttering up with messages that are inopportune in this state. Process `ml_filter` only discards messages from previous sessions.

The Communicating State — Lower Part

When the OM-protocol entity resides in the communicating state, it has to make sure no OM PDU gets lost. Actually, this state is easy to specify. Again a distinction is made concerning the roles of Manager and Agent. Informally, the events that can occur for the Manager are as follows:

- Every PDU, communicated over gate `CON`, with a `req` parameter, will be sent to the peer entity using an invoke request service primitive, carrying the current session-identifier.
- Every PDU, communicated over gate `UNC` will be sent to the peer entity using an invoke request service primitive, carrying the current session-identifier.
- When a response indication is communicated over gate `RSAP`, carrying an OM PDU and a session-identifier equal to the current session-identifier, the OM PDU is communicated over gate `CON` with a `conf` parameter.

- When an eventreport indication is communicated over gate `RSAP`, carrying an OM PDU and a session-identifier equal to the current session-identifier, the OM PDU is communicated over gate `MNS`.
- Invoke indications, communicated over gate `RSAP`, carrying session-identifiers, other than the current session-identifier, and an OM PDU, other than an enable PDU, are rejected using the reject request service primitive.
- Reject-, response- and eventreport indications, communicated over gate `RSAP`, carrying a session-identifier other than the current session-identifier, are discarded.

This concludes the brief informal description for the communicating state, dealing only with opportune messages. The inopportune messages are dealt with in the aborting state, so these are orthogonal to the events discussed above. A formal discription of the above described scheme is given in process `ml_communicate`.

```
PROCESS ml_communicate [RSAP,CON,UNC,MNS] (aa: AId, n: AI, j: nat) : NOEXIT :=

  ml_con [RSAP,CON] (aa,n,j)
|||
  ml_unc [RSAP,UNC] (aa,n,j)
|||
  ml_mns [RSAP,MNS] (aa,n,j)
|||
  ml_rejectold [RSAP] (aa,n,j)

WHERE

  PROCESS ml_con [RSAP,CON] (aa: AId, n:  AI, j: nat) : NOEXIT :=

    CON !aa !get !req;
    RSAP !n !invokereq !initiator(aa) !responder(aa) !get !aa !j;
    STOP
 |||
    RSAP !n !resultind !responder(aa) !initiator(aa) !get !aa !j;
    CON !aa !get !conf;
    STOP

 ENDPROC (* ml_con *)

 PROCESS ml_unc [RSAP,UNC] (aa: AId, n: AI, j: nat) : NOEXIT :=

   UNC !aa !uset;
   RSAP !n !invokereq !initiator(aa) !responder(aa) !uset !aa !j;
   STOP

 ENDPROC (* ml_unc *)

 PROCESS ml_mns [RSAP,MNS] (aa: AId, n: AI, j: nat) : NOEXIT :=

   RSAP !n !eventreportind !responder(aa) !initiator(aa) !report !aa !j;
   MNS !aa !report;
   STOP

 ENDPROC (* ml_mns *)

 PROCESS ml_rejectold [RSAP] (aa: AId, n: AI, j: nat) : NOEXIT :=
      RSAP !n !invokeind !responder(aa) !initiator(aa) ?p : OMPDU !aa ?m : nat
           [not(m eq j) and not(IsEnable(p))];
      RSAP !n !rejectreq !initiator(aa) !responder(aa) !p !aa !m;
      ml_rejectold [RSAP] (aa,n,j)
    []
      RSAP !n ?s : RSP !responder(aa) !initiator(aa) ?p : OMPDU !aa ?m : nat
           [not(m eq j) and not(IsInvokeInd(s)) and IsInd(s)];
      ml_rejectold [RSAP] (aa,n,j)

    ENDPROC (* ml_rejectold *)

  ENDPROC (* ml_communicate *)
```

Again, some differences exist in the events that can occur in the communicating state for the Agent side of the association. Informally, these events can be described as follows:

- When an invoke indication, carrying an OM PDU and a session-identifier, equal to the current session-identifier, the OM PDU is communicated either over gate `CON`, when the PDU is a `get` PDU (or in general, represents a user confirmed service element), or over gate `UNC`, when the PDU is an `uset` PDU.
- Every PDU communicated over gate `MNS` will be sent to the peer entity using the eventreport request service primitive, carrying the current session-identifier.
- Invoke indications, communicated over gate `RSAP`, carrying session-identifiers, other than the current session-identifier, and an OM PDU, other than an enable PDU, are rejected using the reject request service primitive.
- Reject-, response- and eventreport indications, communicated over gate `RSAP`, carrying a session-identifier other than the current session-identifier, are discarded.

These events are formally expressed by process `al_communicate`.

```
  PROCESS al_communicate [RSAP,CON,UNC,MNS] (aa: AId, n: AI, j : nat) :NOEXIT :=

    al_con [RSAP,CON] (aa,n,j)
  |||
    al_unc [RSAP,UNC] (aa,n,j)
  |||
    al_mns [RSAP,MNS] (aa,n,j)
  |||
    al_rejectold [RSAP] (aa,n,j)

  WHERE

    PROCESS al_con [RSAP,CON] (aa: AId, n: AI, j: nat) : NOEXIT :=

      RSAP !n !invokeind !initiator(aa) !responder(aa) !get !aa !j;
      CON !aa !get !ind;
      STOP
    |||
      CON !aa !get !res;
      RSAP !n !resultreq !responder(aa) !initiator(aa) !get !aa !j;
      STOP

    ENDPROC (* al_con *)

    PROCESS al_unc [RSAP,UNC] (aa: AId, n : AI, j: nat) : NOEXIT :=
      RSAP !n !invokeind !initiator(aa) !responder(aa) !uset !aa !j;
      UNC !aa !uset;
      STOP

    ENDPROC (* al_unc *)

    PROCESS al_mns [RSAP,MNS] (aa: AId, n: AI, j: nat) : NOEXIT :=

      MNS !aa !report;
      RSAP !n !eventreportreq !responder(aa) !initiator(aa) !report !aa !j;
      STOP

    ENDPROC (* al_unc *)


    PROCESS al_rejectold [RSAP] (aa: AId, n: AI, j: nat) : NOEXIT :=

      RSAP !n !invokeind !initiator(aa) !responder(aa) ?p : OMPDU !aa ?m : nat
           [not(m eq j) and not(IsEnable(p))];
      RSAP !n !rejectreq !responder(aa) !initiator(aa) !p !aa !m;
      al_rejectold [RSAP] (aa,n,j)
    []
      RSAP !n ?s : RSP !initiator(aa) !responder(aa) ?p : OMPDU !aa ?m : nat
           [not(m eq j) and not(IsInvokeInd(s)) and IsInd(s)];
      al_rejectold [RSAP] (aa,n,j)

    ENDPROC (* al_rejectold *)

  ENDPROC (* al_communicate *)
```

This concludes the Lower Protocol Functions for both the Manager and the Agent side of the association in the communicating state. Note that in this state it was decided that the messages communicated over gate `RSAP`, carrying session-identifiers other than the current session-identifier are to be discarded. Deciding to accept for instance the old eventreports would have rendered a different service than the one offered here!

The Releasing state — Lower part

The Lower Protocol Functions for the releasing state are fairly easy to specify, since there is only one PDU that affects this state: a disable PDU. The Manager and Agent side of the association are almost equivalent in their functionality in this state. The informal description for the Manager side of the association is as follows:

- When a disable PDU is communicated at the `REL` gate, this PDU is sent to the peer entity using an invoke request carrying the current session-identifier. Subsequently, all communication over gates `MNS` and `UNC` is blocked. Only communication over gate `CON` is still allowed.

Note that when the Manager issues the disable request, there may still be PDUs that can be classified as user confirmed, that can be sent, even after the disable request was sent. However, no new requests can be added, since the Upper Protocol Functions do not allow for this kind of communication to take place anymore. The formal description in this state for the Manager side of the association is specified by process `ml_release`.

```
  PROCESS ml_release [RSAP,CON,UNC,MNS,REL] (aa: AId, n: AI, j : nat) : NOEXIT :=

    CON !aa ?p : OMPDU ?k : KIND;
```

```
      ml_release [RSAP,CON,UNC,MNS,REL] (aa,n,j)
    []
      UNC !aa ?p : OMPDU;
      ml_release [RSAP,CON,UNC,MNS,REL] (aa,n,j)
    []
      MNS !aa ?p : OMPDU;
      ml_release [RSAP,CON,UNC,MNS,REL] (aa,n,j)
    []
      REL !aa !disable;
      RSAP !n !invokereq !initiator(aa) !responder(aa) !disable !aa !j;
      ml_releasing [RSAP,CON] (aa,n,j)

    WHERE

      PROCESS ml_releasing [RSAP,CON] (aa: AId, n: AI, j: nat) : NOEXIT :=

        CON !aa ?p : OMPDU ?k : KIND;
        ml_releasing [RSAP,CON] (aa,n,j)
      []
        RSAP !n !invokeind !responder(aa) !initiator(aa) ?p : OMPDU !aa !j
             [not(IsDisable(p))];
        RSAP !n !rejectreq !initiator(aa) !responder(aa) !p !aa !j;
        ml_releasing [RSAP,CON] (aa,n,j)
      []
        RSAP !n !resultind !responder(aa) !initiator(aa) ?p : OMPDU !aa !j
             [not(IsDisable(p))];
        ml_releasing [RSAP,CON] (aa,n,j)
      []
        RSAP !n !eventreportind !responder(aa) !initiator(aa) !report !aa !j;
        ml_releasing [RSAP,CON] (aa,n,j)

      ENDPROC (* ml_releasing *)

    ENDPROC (* ml_release *)
```

The process representing Lower Protocol Functions for the Agent side of the association in the releasing state is (as stated before) almost identical to that for the Manager. The informal description is as follows:

- When a disable PDU is communicated at the REL gate, this PDU is sent to the peer entity using an invoke request carrying the current session-identifier. Subsequently, all communication over gates MNS and UNC and CON is blocked.

Note that the possible responses for previously received indications of user confirmed service primitives cannot be sent once a disable PDU is sent.

```
    PROCESS al_release [RSAP,CON,UNC,MNS,REL] (aa: AId, n: AI, j: nat) : NOEXIT :=

      CON !aa ?p : OMPDU ?k : KIND;
      al_release [RSAP,CON,UNC,MNS,REL] (aa,n,j)
    []
      UNC !aa ?p : OMPDU;
      al_release [RSAP,CON,UNC,MNS,REL] (aa,n,j)
    []
      MNS !aa ?p : OMPDU;
      al_release [RSAP,CON,UNC,MNS,REL] (aa,n,j)
    []
      REL !aa !disable;
      RSAP !n !invokereq !responder(aa) !initiator(aa) !disable !aa !j;
      al_releasing[RSAP](aa,n,j)

    WHERE

      PROCESS al_releasing [RSAP] (aa: AId, n: AI, j: nat) : NOEXIT :=

        RSAP !n !invokeind !initiator(aa) !responder(aa) ?p : OMPDU !aa !j
             [not(IsDisable(p))];
        RSAP !n !rejectreq !responder(aa) !initiator(aa) !p !aa !j;
        al_releasing [RSAP] (aa,n,j)
      []
        RSAP !n !resultind !initiator(aa) !responder(aa) ?p : OMPDU !aa !j
             [not(IsDisable(p))];
        al_releasing [RSAP] (aa,n,j)
      []
        RSAP !n !eventreportind !initiator(aa) !responder(aa) !report !aa !j;
        al_releasing [RSAP] (aa,n,j)

      ENDPROC (* al_releasing *)

    ENDPROC (* al_release *)
```

Again, processes `al_release` and `ml_release` do not block communication at the gate `RSAP`. This is done to insure progress.

The Aborting State — Lower Part

The final state is represented by the aborting state. The events that can occur in this state are orthogonal to those in the communicating state and the releasing state. The process representing this state is the same for both the Manager and the Agent side of the association. Informally, the following events are specified:

- When an invoke indication is communicated at the `RSAP` gate, carrying a disable PDU and a session-identifier equal to the current session-identifier, a reply is sent using the response request to carry the PDU. The PDU itself is communicated at the `ABORT` gate, parameterized with an `ind`.
- When a reject indication is communicated at the `RSAP` gate, carrying a session-identifier equal to the current session-identifier, this is interpretted as the fact that the peer is "out of sync" with the entity itself, since it has rejected a previous message. A disable PDU is communicated over gate `ABORT`, parameterized with an `ind`.
- When a result indication is communicated at the `RSAP` gate, carrying a disable PDU and a session-identifier equal to the current session-identifier, this PDU is mapped onto the `ABORT` gate, parameterized with a `conf`.
- When an invoke indication is communicated at the `RSAP` gate, carrying an enable PDU and an arbitrary session-identifier, a reply is constructed using the reject request to carry the PDU. Furthermore, a disable PDU is communicated over gate `ABORT`, parameterized with and `ind`, to reset the entity to the initial state. This is done since the peer entity is "out of sync" with the entity receiving the PDU.

The above described events are formally described by process `l_abort`.

```
    PROCESS l_abort[RSAP,ABORT] (aa: AId, n: AI, j: nat) : EXIT :=

      RSAP !n !rejectind !responder(aa) !initiator(aa) ?p : OMPDU !aa !j;
      ABORT !aa !disable !ind;
      EXIT
    []
      RSAP !n !resultind !responder(aa) !initiator(aa) !disable !aa !j;
      ABORT !aa !disable !conf;
      EXIT
    []
      RSAP !n !invokeind !responder(aa) !initiator(aa) !disable !aa !j;
      RSAP !n !resultreq !initiator(aa) !responder(aa) !disable !aa !j;
      ABORT !aa !disable !ind;
      EXIT
    []
      RSAP !n !invokeind !responder(aa) !initiator(aa) !enable !aa ?m : nat;
      RSAP !n !rejectreq !initiator(aa) !responder(aa) !enable !aa !m;
      ABORT !aa !disable !ind;
      EXIT

    ENDPROC (* l_abort *)

  ENDPROC (* association *)

ENDSPEC
```

4.4 Summary

In this Chapter, the OM protocol entity model is presented, formally described using LOTOS and the decomposition techniques described in Section 4.2. Note that again several simplifications have been made with respect to the specification found in Appendix B, most notably the following:

- The number of communications is reduced.
- The number of associations an OM-protocol entity can handle is reduced to only one.

These simplifications were made to meet the demands CÆSAR poses on a LOTOS specification (see also Chapter 2 and 3).

Chapter 5

Tools and Validation

The main issues, addressed in this Chapter, are the validation of the three models presented in the previous Chapters and the results thereof. The validation was performed using two toolboxes, LITE [Eer95] and EUCALYPTUS [Gar96]. A brief introduction on these toolboxes will be given in Section 5.1. Validation using LITE will be described in Section 5.2 and the validation using EUCALYPTUS will be discussed in Section 5.3.

5.1 Toolboxes

The toolboxes used for the validation are LITE and EUCALYPTUS. Both toolboxes offer a wide range of utilities to aid the user in understanding a LOTOS-specification, and proving properties of the specifications.

The toolbox LITE harbours some advanced techniques such as *goal-oriented* execution and transformation of specifications into *Extended Finite State Machines*. Furthermore, the interactive simulation of LOTOS-specifications is possible, and the LOTOS-specification can be checked both syntactically and static-semantically. A graphical representations of a LOTOS-description can automatically be drawn as well. Several other utilities are offered for LITE, however, the version available at CMG was only a limited version, offering not all tools that usually come with the toolbox.

The toolbox EUCALYPTUS offers a wide range of functionalities, including simulation, compilation, verification and test case generation for LOTOS descriptions. The verification includes verifying temporal logic formulas, and proving bisimulation-equivalences between two LOTOS-descriptions. The version of this toolbox, available at CMG does not offer the complete set of programs that come with the toolbox. However, the important tools, such as CÆSAR, CÆSAR.ADT and ALDEBARAN are supported. These tools provide the means for simulation, validation and generating *Labeled Transition Systems* (LTSs).

5.2 Using LITE

The models presented in Chapters 2, 3 and 4, were modelled after initial specifications for the OM-service, the RR-service and the OM-protocol entity. These initial specifications, described in full-LOTOS, represented the information found in [AVV97] and some additional requirements (see Appendices A and B for the specifications and the requirements).

A first step in the validation of these specifications was to check the syntax and the static semantics of the LOTOS-descriptions. LITE was used to check for this automatically. As was expected, this revealed several errors in the specifications, which had to be corrected. The most recurring errors were those where the functionalities of processes was incorrect (i.e. `EXIT` which should be `NOEXIT` and vice versa).

Subsequently, after the correction of these errors, *single step* simulation of the OM-service specification using SMILE, was used to obtain an initial indication on the correctness of the specified system. The single step mechanism allows one to evaluate all possible next events starting at a given event.
This again resulted in the rewriting of several processes, most of them dealing with the synchronizing end-to-end processes, which proved either too restrictive or introduced deadlocks.

When finally, the OM-service specification was believed to be correct (with respect to its behaviour), several test cases were made to strengthen this believe. Some theoretical reflections on test cases are discussed in Section 5.2.1. Subsequently, Section 5.2.2 describes the actual simulation of these test cases using LITE.

5.2.1 Different Classes of Test Cases

When considering test cases, a distinction between two classes of test cases can be made. A test case can be classified either as a *may* test case, or a *must* test case. In general, the may test cases are more restrictive than the must test cases: when a may test case is run in parallel with a specification, deadlocks can be produced, but *at least one* trace must successfully terminate for the specification to adhere to the test case. May test cases usually specify a behaviour that a specification must exhibit under *certain* conditions.
The must test cases are usually more general than the specification that is to be tested, and as such no deadlock may occur when run in parallel with a specification.

The following example will clarify the distinction between these two classes of test cases. The example is taken from [BW90], and although there it is used to show the difference between two processes, it can also be used to show the difference between the two classes of test cases.

Example The example describes the following situation: you find yourself in a building with two elevators, one of which is perfectly safe, but the other definitely is not. In the hall you find two elevator doors, but unfortunately you cannot tell which one leads to the safe elevator. Now you call for the elevator by pushing the "up"-button and after a while a door opens ...
Using the following atomic actions we can describe this situation:

- `DOOR` = the elevator doors open,
- `RISK` = the offered elevator is highly dangerous,
- `SAFE` = the offered elevator is perfectly safe.

The above sketched situation, can be described by the following LOTOS-description (see also figure 5.1):

```
  DOOR; RISK; EXIT
[]
  DOOR; SAFE; EXIT
```

Suppose one of the properties that is to be tested, is whether this description includes a scenario that involves using the elevator and going up in a safe way. This test case can be described by the following LOTOS-description (see also figure 5.2):

```
  DOOR; SAFE; EXIT
```

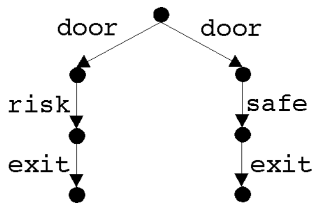

Figure 5.1: *The elevator dilemma: an example*

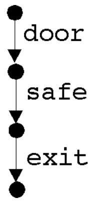

Figure 5.2: *A may test case*

The test case, depicted in figure 5.2, is a typical example of a may testcase: it describes only the situation in which, by chance perhaps, the safe elevator was chosen, but the option that describes that the dangerous elevator is chosen is discarded. When this test case is run in parallel with the description represented by figure 5.1, the execution sequences reveal one deadlocking trace (due to the non-determinism in the process represented by figure 5.1):

```
DOOR; STOP
```

and one successfully terminating trace:

```
DOOR; SAFE; EXIT
```

Now suppose you volunteer to test which of the elevators is safe and which is dangerous. Once the doors open, the elevator offered is either safe or dangerous, and you will find out by getting in only. This can be described by the following LOTOS-description (see also figure 5.3):

```
DOOR;
( SAFE; EXIT
[]
  RISK; EXIT )
```

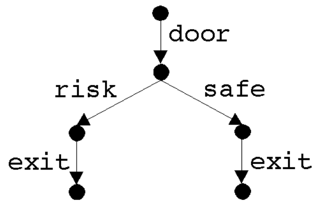

Figure 5.3: *A must test case*

The test case, depicted in figure 5.3, is a typical example of a must test case: the non-determinism in the process represented by figure 5.1 is taken into account. When this test case is run in parallel with the description represented by figure 5.1, the execution sequences reveal no deadlocking traces and two successfully terminating traces:

```
DOOR; SAFE; EXIT
```

and

```
DOOR; RISK; EXIT
```

Although must test cases are preferable with regard to deadlock detection and exhaustive testing, these test cases are very hard to specify. This usually has to do with both the non-determinism in, and the magnitude of the specification that is to be tested. May test cases, however, are much easier to specify, since they only highlight one aspect of the specification that is to be tested.

5.2.2 Validating Specifications using Lite

After initial simulation was performed using the tool Smile, part of the toolbox Lite, several test cases were constructed (see Appendix C) that each specified desired properties the OM-service specification should exhibit. These test cases can be classified as may test cases, since every test case highlights only one aspect of the OM-service specification, and not the overall behaviour of the OM-service specification.
It must be noted that these test cases were not formally derived, but they were inspired by the view the Managed Objects have on the states an association can be in (see figure 2.6). Every state is at least once visited by at least one test case.

The specified test cases were first run in parallel with the OM-service specification, using the **Run Test** command, which allows to execute the parallel composition of a test case and the specification. Smile offers several options for performing simulations of this kind:

- Goal oriented execution.
- Expansion up to a pre-defined unfold depth.
- Single step expansion.

Goal oriented execution and expansion. The goal oriented execution and the expansion up to a certain depth can be performed breadth-first or depth-first. Using the goal oriented execution, one must mark an event in the Lotos-description as the target. Next, Smile tries to execute the specification in such a way that the target behaviour expression is reached. Not all events are computed: events definitely not leading to the goal are omitted, and as such, only partial expansion takes place.
The expansion up to a predefined unfold depth is a complete expansion: every event is evaluated until the predefined depth is reached.

Unfortunately, on a specification with a large data component (such as in the OM-service specification), the two above described methods do not lead to satisfactory results: the computation time is long and runs out of memory before the results are available. Most of this has to do with the evaluation of the Abstract Data Types: finding solutions to predicates takes, if even possible, very long and is very memory consuming.

Single step expansion. Another option for performing the simulation of the test cases is by single stepping through the specification. Using this method, one is able to selectively expand the specification, which both reduces the amount of time spent and the amount of memory used in the computation. Selecting the right events to expand one step is aided by the option **Find Action Prefix**. This option shows which action prefix expression (or the process in which the action prefix expression occurs) is defined, and thus enables one to see which processes participate in the event. This option also helps in getting a more profound understanding of the specification.

The test cases revealed that several modifications had to be made to the OM-service specification. The OM-service specification did not adhere to all the test cases. Again the problem was mostly found in the end-to-end constraints, which were too restrictive. After several corrections, finally the OM-service specification adhered to the test cases.
Subsequently these test cases were used again for the OM-protocol. Again, the test cases revealed several misconceptions in the OM-protocol, which resulted in modifications for the OM-protocol. Although the OM-protocol was in essence harder to specify, less errors were found. This probably was due to the following three reasons:

1. A gained insight in the specification styles.
2. A gained insight in the specification formalism.
3. The already established framework in which to specify the OM-protocol entity (i.e. the local constraints in the OM-service specification).

Finally, both OM-service and OM-protocol adhered to the test cases, and were considered to be ready for further validation and analysis.
An attempt was made to build an Extended Finite State Machine for the OM-service and the OM-protocol, however, this failed. The reasons for this were again the amount of memory and time needed for the computations.
The possibilities of LITE for further validation are limited: for instance no deadlock detection and livelock detection is supported. To this end, the toolbox EUCALYPTUS was used.

5.3 Using EUCALYPTUS

The toolbox EUCALYPTUS offers several tools for analyzing and validating a LOTOS-description. However, a major drawback of the tools that come with EUCALYPTUS is the restrictions they pose on the LOTOS-descriptions: not the full set of LOTOS-operators is supported. In order to be able to perform deadlock detection, livelock detection, etc. on the specifications, the specification has to be simplified. Thus, not the specification itself is validated, but a *model* of the specification is validated. Remarks on the simplifications made for the OM-service and the OM-protocol are discussed in Section 5.3.1. A discussion on the use of the tools and the results is given in Section 5.3.2.

5.3.1 Some Notes on the Simplifications

In order to analyse the specifications using EUCALYPTUS, several simplifications had to be made. Most of these simplifications dealt with the rewriting of LOTOS-descriptions, since EUCALYPTUS only accepts a subset of the LOTOS-language. This, of course, can have its impact on the dynamic behaviour of any specification: properties get lost or are modified. The restrictions on the LOTOS-operators are the following: no process recursion is allowed on the left and right hand part of the parallel-operator `|[...]|`, nor on the left hand part of the enable-operator `>>` and the disable operator `[>`. Of these restrictions, the restrictions on the parallel-operator have the most impact for the OM-service specification and the OM-protocol specification. For instance, the message reordering in the OM-service specification is specified using the interleaving operator `|||`, a special case of the parallel-operator `|[...]|`.

For the OM-service specification, as described in Appendix A, there exist two problem-areas:

- The number of concurrent associations, modelled in the OM-service specification using the interleaving operator in combination with process recursion.

- The number of messages in transit at one point in time, also modelled using the interleaving operator in combination with process recursion.

It was decided to model the (possibly infinite) number of concurrent associations with only one association. This simplification is not seen as too restrictive, as each association is modelled as an independent process, and has a unique identifier that is used to address this association. Thus it is unlikely that one association influences another association.
A second simplification that was made concerns the number of messages that can be in transit at one point in time. It was decided that only three messages, one user confirmed and two unconfirmed messages are modelled. Again, this simplification is not considered to be too restrictive: part of the message reordering is still present in the model, since for instance one unconfirmed message can still "overtake" a user confirmed message. Furthermore, it is not expected that the messages communicated in the communicating state present problems with respect to the state transition from the communicating state to the releasing state or the aborting state.

An additional problem was encountered for the ADT `OM_Service_Primitives`, which could not be dealt with by CÆSAR. The reason for this problem (also mentioned on page 2.3.1), was found out by trial-and-error. The solution to this problem, i.e. rewriting the ADT `OM_Service_Primitives` and using the polymorphic gate features of LOTOS, involved the rewriting of all LOTOS-descriptions using the sort `OSP`.

The OM-protocol entity has, besides the problems mentioned for the OM-service, some additional problem-areas:

- The underlying service, i.e. the RR-service specification which specifies a (possibly infinite) number of messages in transit, modelled using the interleaving operator.
- The number of sessions for one association, which can be infinite: each session is identified using a unique natural number.

With respect to the underlying service: this is restricted to a finite number of messages, with a maximum of four, that can be in transit at one point in time. Allowing for more messages would yield LTSs that exceed the limits of computer memory.

Surprisingly, the number of sessions is one of the items that has to be reduced as well. This is due to the restrictions that are imposed on the datapart of a LOTOS-description: only a finite subset of the natural numbers can be used. As such, the current session-identifier, used to distinguish the current session from previous (or future) sessions can only be chosen from a finite set. Two options exist: reuse used session-identifiers or considering only a finite number of sessions. For simplicity, the latter option is chosen.

5.3.2 Validating Specifications using EUCALYPTUS

The toolbox EUCALYPTUS offers several tools. Each of these tools has its own functionalities. The tools used mainly on the models described in Chapters 2, 3 and 4 are the following:

- CÆSAR
- CÆSAR.ADT
- ALDEBARAN

The tools CÆSAR and CÆSAR.ADT are both used to compile the LOTOS-specifications into Labeled Transition Systems (LTSs). First, the specification is syntactically and static-semantically checked. Then the specification is checked to be in accordance with the restrictions imposed on the static and dynamic parts of the LOTOS language. Finally an LTS is built. These LTSs are then used to

further analyse and validate the specification. This analysis and validation is performed mainly by ALDEBARAN.
The tool ALDEBARAN is used to check the LTSs for deadlock and livelock. Other options are to reduce the LTSs, generated by CÆSAR and CÆSAR.ADT modulo some equivalence relation (strong, observational, etc.).

5.3.3 Validation of the OM-service model

In order to validate the OM-service model, described in Chapter 2, an LTS has to be generated using CÆSAR and CÆSAR.ADT. To this end, two approaches can be used:

1. generate the LTS representing the OM-service model at once (i.e. non-compositionally).
2. generate the LTS representing the OM-service model compositionally.

The non-compositional approach. Although the OM-service model was considered to be relatively small, during the generation of the LTS in one run, it was found out that this approach was a long running process. More than three hours of computing time were necessary to finish the LTS. The computation of an LTS that is equivalent, modulo strong equivalence, to the OM-service model LTS could not be completed due to the limits of memory. Statistics of this approach can be found in table 5.1.

Process	# States	# Transitions	# τ Transitions	Reduction Modulo
`association`	420611	2404442	n.c	none
	n.c.	n.c.	n.c.	strong equivalence

Table 5.1: *Statistics for the OM-service model* non-compositional *approach*
The n.c. means that this could not be computed

The compositional approach. A second option is to generate the LTS for the OM-service model using a compositional strategy. This strategy enables one to generate LTSs for individual processes which in turn can be combined. To illustrate: the process `association` is a parallel composition of three distinct processes:

```
PROCESS association [OSAP] (aa : aid) : NOEXIT :=

  (agent [OSAP] (aa)  ||| manager [OSAP] (aa) ) || sync [OSAP] (aa)

ENDPROC (* association *)
```

The tools CÆSAR and CÆSAR.ADT are used to generate LTSs for the processes `agent`, `manager` and `sync`. Next, these LTSs can be reduced modulo strong equivalence, yielding new LTSs. Assuming the file containing the reduced LTS for process `agent` is agent.aut, the reduced LTS for process `manager` is contained in file manager.aut and the reduced LTS for process `sync` is contained in file sync.aut. These LTSs can then be combined by generating the LTS for a file service.exp, containing the following information:

```
BEHAVIOUR  (agent ||| manager) || sync
```

The LTS thus created represents process `association`, and can be reduced modulo strong equivalence, yielding an LTS that is free of deadlock and free of livelock. Table 5.2 contains the statistics for the LTSs thus created.

Process	# States	# Transitions	# τ Transitions	Reduction Modulo
`sync`	3889	45801	5184	none
	2700	31800	3240	strong equivalence
`agent`	39	389	4	none
	16	187	2	strong equivalence
`manager`	46	497	4	none
	18	201	2	strong equivalence
`association`	97528	492714	150948	none
	52538	283512	80720	strong equivalence

Table 5.2: *Statistics for the OM-service model,* compositional *approach*

Previous models for the OM-service, checked using this approach, contained some unwanted behaviour with respect to the too strict separation between the establishing and the relinquishing of an association. This was found out by interactive simulation of the service model using the tool XSIMULATOR. This resulted in several adaptions of the process `pcsync` and finally resulted in the OM-service model as described in Chapter 2.

Compared to the non-compositional approach, the compositional approach is much faster. To illustrate: the calculation of the OM-service model using the non-compositional approach took more than three hours, whereas the calculation of the OM-service model using the compositional approach all in all took less than 30 minutes.

5.3.4 Validation of the OM-protocol model

The OM-protocol model, validated in this Section, is described by the following LOTOS specification:

```
SPECIFICATION OM_Protocol [OSAP] : NOEXIT

BEHAVIOUR

  HIDE RSAP IN (
  ( OM_Protocol_Entity [OSAP, RSAP] (aid(offset,neighbour),offset)
  |||
    OM_Protocol_Entity [OSAP, RSAP] (aid(offset,neighbour),neighbour) )
  |[RSAP]|
    RR_Service [RSAP]
  )

ENDSPEC (* OM_Protocol *)
```

This process uses the specification for the OM-protocol entity model as described in Chapter 4, and the specification for the lower-layer service, the RR-service model, described in Chapter 3.

In order to check this OM-protocol model for absence of deadlock and livelock, adjustments have to be made to two processes described in Chapters 2 and 4. These adaptions are introduced to accomodate for the maximal number of sessions that are considered. First, the process `manager` is adapted, to disallow the Manager from starting more than one session. The process that subsequently is used is the following:

```
  PROCESS manager[OSAP](aa : aid, j : nat) : NOEXIT :=

    [j lt succ(0)] -> (
       minitiate [OSAP] (aa)
      >>
         (((mcommunicate [OSAP] (aa) || initiaterelease [OSAP] (initiator(aa)) )
          [>
         forcerelease [OSAP] (initiator(aa))  )
         ||
        syncdisable [OSAP] (initiator(aa)) )
         >>
        manager [OSAP] (aa, succ(j))
         )
     []
       [j eq succ(0)] -> (i; manager [OSAP] (aa,j) )

  ENDPROC (*manager*)
```

The initial call for process `manager` in process `association` in the OM-protocol entity model, is changed from `manager[OSAP] (aa)` into `manager[OSAP] (aa,0)`.

Another restriction that is imposed on the OM-protocol, is the restriction of the lower layer service. Process `rrsi`, used in the OM-protocol entity, which represents the local interface with the RR-service, is constrained as follows:

```
PROCESS rrsi [RSAP] (n : AI, aa: AId) : NOEXIT :=

  RSAP !n ?s : RSP ?k : AI ?l : AI ?p : OMPDU !aa !0
       [not(k eq l) and (IsEventreport(s) eq Isreport(p))];
  rrsi [RSAP] (n, aa)

ENDPROC (* rrsi *)
```

This process is adapted to allow for communication over gate `RSAP` constrained to one session-identifier only. Furthermore, a strict relation on several gate parameters is enforced, in order to generate an LTS as small as possible. If these constraints are omitted, the LTS that has to be generated will be beyond the scope of computation time (and memory).

The Compositional Approach

The number of states represented by the OM-protocol is still too large to deal with in one piece: the generation of an LTS is beyond the capabilities of CÆSAR within the limits of memory and time. Fortunately, CÆSAR allows for compositional generation of LTSs. In the case of the OM-protocol, three processes can be distinguished:

- One process represents the lower layer service, the RR-service.
- One process represents the Agent side of the association (i.e. the Agent OM-protocol Entity (APE)).
- One process represents the Manager side of the association (i.e. the Manager OM-protocol entity (MPE)).

The RR-service model. The RR-service model is restricted to at most four messages that can be in transit at any point in time. Furthermore, it is restricted by running it in parallel with process `rrsi`, which represents the interface with an OM-protocol entity. The generation of the LTS representing this RR-service model using the non-compositional approach, however, is still very time-consuming.

Fortunately, the RR-service model can be dealt with compositionally as well, which decreases the amount of time needed vastly. First an LTS is generated for process `communication`, which represents only one communication. The LTS thus generated is reduced modulo strong equivalence. Next, this reduced LTS is interleaved with itself, yielding an LTS representing at most two communications that can be in transit at any point in time. Again, this LTS is reduced modulo strong equivalence. This is repeated until a reduced LTS for four messages in transit is obtained.

Absence of deadlock and livelock was validated using ALDEBARAN. The statistics with respect to the LTSs generated can be found in table 5.3.

# Messages in Transit	# States	# Transitions	# τ Transitions	Reduction Modulo
one	53	728	0	none
	27	52	0	strong equivalence
two	729	2808	0	none
	378	1404	0	strong equivalence
three	10206	57564	0	none
	3654	19656	0	strong equivalence
four	98658	720720	0	none
	27405	190008	0	strong equivalence

Table 5.3: *Statistics for the RR-service model,* compositional *approach*

Compared to the non-compositional approach, again the compositional approach is much faster: the results presented above can be calculated within 30 minutes, whereas the calculation using the non-compositional approach was aborted after 13 hours, since no progress had been noticed for 8 hours in a row.

The Manager OM-protocol entity. The problems encountered for the RR-service layer repeated themselves using the non-compositional approach for the Manager OM-protocol entity. Again, the compositional approach was endeavored. An LTS was generated for the Upper Protocol Functions, and one LTS was generated for the Lower Protocol Functions. These LTSs were reduced modulo strong equivalence and subsequently combined into a new LTS. This LTS in turn, representing the MPE, was reduced modulo strong equivalence. Statistics can be found in table 5.4.

Process	# States	# Transitions	# τ Transitions	Reduction Modulo
Upper Protocol Functions	408	1551	25	none
	144	692	10	strong equivalence
Lower Protocol Functions	13495	186762	45	none
	290	2941	2	strong equivalence
MPE	1749	13571	1082	none
	860	7973	100	strong equivalence

Table 5.4: *Statistics for the Manager OM-protocol entity model,* compositional *approach*

The Agent OM-protocol entity. For the Agent OM-protocol entity, the compositional approach is used as well. An LTS was generated for the Upper Protocol Functions, and one LTS was generated for the Lower Protocol Functions. These LTSs were reduced modulo strong equivalence and subsequently combined into a new LTS. This LTS in turn, representing the APE, was reduced modulo strong equivalence. Statistics can be found in table 5.5.

The calculation of the Lower Protocol Functions were clearly harder than the Upper Protocol Functions. For instance, the calculation of the Lower Protocol Functions for the MPE took more than 30 minutes, whereas the Upper Protocol Functions for the MPE took less than 30 seconds. Combining these two LTSs is a matter of minutes. Clearly, using the non-compositional approach

Process	# States	# Transitions	# τ Transitions	Reduction Modulo
Upper Protocol Functions	256	809	25	none
	89	370	18	strong equivalence
Lower Protocol Functions	3922	51283	15	none
	175	1790	2	strong equivalence
APE	796	5285	498	none
	377	2875	331	strong equivalence

Table 5.5: *Statistics for the Agent OM-protocol entity model,* compositional *approach*

would have taken much longer.

The OM-protocol model. Finally, the three LTSs were combined, yielding one LTS, representing the OM-protocol model. This model was reduced modulo strong equivalence. Statistics concerning these LTSs can be found in table 5.6. The OM-protocol model, thus created, was found not to be free of deadlock. A trace, containing the deadlock was analyzed and revealed that a request for a fifth message caused the deadlock. This was cured by adding a fifth message that can be in transit to the RR-service model.

Process	# States	# Transitions	# τ Transitions	Reduction Modulo
OM-Protocol	47525	209190	25238	none
	5180	21272	14942	strong equivalence

Table 5.6: *Statistics for the OM-protocol model,* compositional *approach*

Considering more than one session. An effort was made to apply the above sketched approach in order to calculate the LTSs for the OM-protocol that represented two sessions. This, however, failed due to the amount of memory needed in this calculation. Considering even more than two sessions therefore seems a goal beyond reality.

5.3.5 The OM-service Model versus the OM-protocol Model

The tool ALDEBARAN offers the possibility to compare two LTSs modulo various equivalence and preorder relations. In order to check whether the OM-protocol is indeed an implementation of the OM-service, the LTSs generated for both models are compared. Since it is unlikely equivalence exists between these two models (the OM-protocol model only describes one session), first an attempt is made to check for various preorder relations. The preorders that can be checked are *strong preorder*, *τ^*a preorder* and *safety preorder* [BFG+91]. According to ALDEBARAN the two latter preorders are equivalent. ALDEBARAN was able to prove that the modulo strong equivalence reduced LTS for the OM-protocol model was indeed a τ^*a preorder *and* a safety preorder for the modulo strong equivalence reduced LTS for the OM-service model. The strong preorder could not be proven.

5.4 Summary

In this Chapter, the validation of the models described in this thesis is discussed. Furthermore, the simulation of the specifications, found in Appendices A and B is highlighted. Several important issues concerning the tools, and the use of the tools were found out:

- In general, tools assist in the understanding of specifications.

- Tools can strengthen the believe in the correctness of a specification.
- The potential of tools is limited in a sense that the state explosion problem is still a problem.

With respect to the state explosion problem it must be noted that the compositional approach, offered in Eucalyptus, does lead to better results.

However, the models, as described in previous Chapters, are severe simplifications of the specification found in the Appendices. As such, it still is vital to know which abstractions can safely be made.

Chapter 6

Concluding Remarks

This Chapter highlights several aspects of both this thesis and the traineeship. Again, this Chapter is divided into several Sections, each dealing with distinct fields. Section 6.1 discusses the documents this thesis is based on. Section 6.2 gives some hints as to how the relationship was between CMG and AVV during the traineeship. Some commentary notes are give on the FDT LOTOS in Section 6.3, together with a discussion on the use of tools in Section 6.4. Finally, Section 6.5 gives an overview for issues for further research.

6.1 Formalizing Informal Documents

This thesis is based on documents written by AVV, and in particular on the document [AVV97]. One of the major omissions in this latter document is, as already mentioned, the lack of specifications concerning the dynamic behaviour of the OM/RR-data communications protocol. This omission formed the starting point for the investigation for this thesis.

Another omission, not noticed until a thorough study was made of [AVV97], is the lack of requirements the OM/RR-protocol has to adhere to. Requirements are felt to be at the heart of both formal specifications and informal specifications for any system. This is because the requirements state what the demands are for the system and what the system's users expects the system to do. As such, these requirements are essential for writing service specifications, arguing about these specifications, validating these specifications and testing implementations for these specifications. Requirements found in [AVV97], and related documents were mostly concerned with the performance aspects of the systems instead of functional requirements. Although important as well, this reflects only a minor part of the requirements needed to design or specify the actual system.

Since no dynamic behaviour was specified in [AVV97] on the one hand, and hardly any requirements regarding the behaviour of the OM/RR-protocol could be found on the other hand, the specifications and the models described in this thesis can only be considered to be a first step towards a real specification. Section 6.5 gives some clues as to what might be changed in both the requirements and the specifications for the system to be according to the needs for the OSS system.

The formalization of a protocol is very hard. If this formalization is to be based on informal documents, the problems encountered usually not so much deal with the "rewriting" of the informal specification into the formal specification. A real problem is often found in the omissions encountered in the informal documents. Formalizing informal documents can thus be regarded as a difficult task, in which many assumptions have to be made.

With respect to the system OSS, a conclusion that can be drawn from this exercise is that the implementation of the system OSS, based on the current documents, has a high risk of failing because of the underspecification found in various documents. Most notably are the underspecification concerning the OM/RR-protocol and the lack of documents relating to the authorization protocol.

6.2 Relation with Respect to AVV

The formal specification for the OM/RR-protocol was to be based on the documents written by AVV, in particular [AVV97]. Soon, it was understood, as mentioned in Section 6.1 that these document alone were not a sufficient basis for this task. Although knowledge of some of the requirements concerning the OM/RR-protocol was available within CMG, several questions remained unanswered.

In order to retrieve this information, a memo containing the more important questions was sent to AVV. Due to management-political reasons, no answer for this memo was received, and as such, several assumptions concerning these questions were made. Although now the specifications were no longer considered to be according to the demands that actually existed on the OM/RR-protocol, it was felt that at least these initial specifications could serve as guidelines to possibly new specifications.

In later stages of the traineeship, feedback by AVV was possible again. An investigative conversation was organized between CMG and AVV concerning the formal specifications of the OM/RR-protocol. The goals for this conversation were set to find out how much of the assumptions that were made concerning the requirements for the OM/RR-protocol, were realistic and/or correct. Subsequently, a list of comments and suggestions on the specifications and the assumptions concerning the requirements as they had been presented to AVV, was received. A second conversation followed, which was more instructive than the first, and more requirements surfaced. Most of these requirements and changes to the current specifications, are discussed in Section 6.5.1.

The conversations with AVV can be characterized as conversations not so much about design and specification, as about implementation. Every so often, remarks were made that steered the discussion into implementation details, and lifting the discussion up to a design level was not always possible. This made it very hard to get to the essence of the matter and retrieve the information.

6.3 The Use of the FDT LOTOS

The FDT LOTOS is found to be very suitable for describing complex systems such as protocol specifications. It offers the possibility for both concise and comprehensive descriptions, and supports the four (major) classes of specification styles. Knowledge about process algebras has to be present, to be able to use LOTOS, and this may form a threshold for potential users.

In relation to ACP it must be mentioned that ACP was found to be more intuitive at first. For instance communication in ACP is, compared to communication in LOTOS, easier to understand. This is because in ACP every communication is explicitly defined using a communication function, in contrast to LOTOS, where communication is synchronised over gates. This makes it hard sometimes to get the overall view of the behaviour of a LOTOS description.

Apart from the few conceptual differences in LOTOS, compared to ACP, some operators are offered in LOTOS, which are not offered in ACP. Most notably is the disable operator `[>`, frequently used in the specifications found in this thesis. This operator is found to be very useful in specifying events that interupt processes that are currently running. Since this is commonly found in data communications protocols, this operator is very effective in modelling real-life situations.

The data part of LOTOS, the Abstract Data Types, was found to be very cumbersome to use. Although a sound formalism, the Abstract Data Types do not add much comfort for the user of the FDT LOTOS. The ADTs as such do not assist in understanding LOTOS descriptions, unlike the dynamic part of LOTOS. As such, a more convenient way of specifying data and functions would be welcome.

Although the industry could very well benefit from the use of Formal Methods in general, the threshold of the FDTs is not lowered by user-unfriendly methods such as Abstract Data Types.

6.4 Tool assistence

This Section is divided into two separate Sections. Section 6.4.1 discusses some issues on tools in general and the tools used in specific, and Section 6.4.2 subsequently discusses the results found by the assistence of tools.

6.4.1 Notes on the Tools

Tool support for simulation and validation of large formal specifications is necessary to gain confidence in the specification. This can be regarded as one of the more important conclusions that can be drawn from this thesis. Actually being able to see that execution sequences exist where a specification is incorrect, or not according to the requirements, strengthens the believe in Formal Methods.
However, there are still some major drawbacks with respect to the use of most tools.

It is still the fact that a (realisticly large) specification has to be simplified to be verified, thus introducing the risk of abstracting too much of the original specification and thereby omitting faulty behaviour (or introducing faulty behaviour).

The use of tools is complicated. Firstly, a profound understanding of the specification language is needed, and even then it is hard to use tools without prior knowledge on these tools. Secondly, the toolboxes, for instance EUCALYPTUS, often have only brief documentation on the functionalities they offer, so obtaining knowledge about these tools is often a case of trial-and-error. The easiest way of learning about these tools (and about the specification language as well) is by starting with very small examples and gradually using bigger examples. A data communications protocol as discussed in this thesis for instance, is clearly too big to learn the elementaries of the toolbox EUCALYPTUS well.
Other drawbacks of tools are concerned with computer memory and computing time. Although nowadays computers have access to an increasing amount of memory, and still gain in computing speed, it is not likely the state explosion problems will be solved in the near future.

Besides these drawbacks, it must be mentioned that tools are man-made. Since people make mistakes, one must always keep in mind that the used tool itself can be incorrect as well. In order to find out about this, a profound understanding of the specification formalism is necessary. This knowledge of the specification formalism is also needed to convince sceptics a specification is correct.

In general, the use of Formal Description Techniques is assisted by the existence of tools. However, these tools are often again as complicated as the formalisms they try to visualize and assist. In the worst case, instead of a preference for Formal Methods, this might even result in an aversion for (the application of) Formal Methods.

6.4.2 Results Found by the Tools

The results found in Chapter 5 using the toolbox Eucalyptus are a bit poor. Partly this is because of the limited amount of time that could be put into the validation excercise. Other reasons, however, are just as significant in this:

1. The computer system harbouring the toolbox Eucalyptus was "limited" in its memory capacity and its availability of time.
2. The documentation that was available for the toolbox Eucalyptus is very brief.
3. The gap that exists between the analyzed LTSs and the Lotos-description.

Especially the third item was slowing down the process of validation. If for instance a deadlock is found in an LTS, the processes that introduce this deadlock cannot be recovered automatically. After modifying the specification, the very time-consuming generation of the LTSs is needed again to check for the absence of the deadlock.

Although a preorder relation was confirmed between the OM-protocol model on the one hand and the OM-service model on the other hand, the ultimate goal was to prove the existence of an equivalence between these two models. Unfortunately, this could not be proven.

6.5 Further Research Topics

Several remarks can be made with respect to the specifications discussed in this thesis. Since these specifications must be regarded as a first effort, it is felt that several changes can be made. These changes, discusses in Section 6.5.1, are motivated by conversations in later stages of the development of the specifications with AVV. Suggestions concerning the tools used in this traineeship, and the use of these tools, are given in Section 6.5.2

6.5.1 Changes to the OM/RR-protocol

In the second conversation with AVV, it became clear that several changes had to be made to the OM/RR-protocol. Although no concrete changes are suggested by AVV, it was felt that the following items are subject to change:

The most notable change would be to model multiple associations on OM-service level, instead of the use of binary associations, assumed in this thesis. To be more specific: a one-to-many association is considered to be the most appropriate for the OM-service layer. In that model, one Agent can be associated to any number of Managers at any time (i.e. one Agent versus a set of Managers). The OM-service layer can then administrate the so-called "subscribers", i.e. a subset of the Managers that have an association with the Agent. This allows for the inclusion of the omitted service elements such as the non-addressed report service and the non-addressed notification service.

The second change concerns the RR-service layer. Instead of the connectionless service it now offers, a connection oriented service will have to be described. With respect to the dynamic behaviour of the RR-service layer, the *Routing Support Agents* (RSAs), described in one of the documents for the OSS, can be taken as a guideline.

Other, conceptual changes are the terminology used in [AVV97] and this thesis. It was found out, that the *Invoker* and *Performer*, described in [AVV97], are equivalent to the Manager and Agent, described in this thesis. The *Initiator* and *Responder* then are terms used for the (connection-oriented) RR-service layer.

Besides these changes, several obscurities remain:

- The notion of subscribers on the level of the RR-service layer is still unclear. It is felt that, with the changes proposed above, the OM-service layer maintains the subscriptions, instead of the RR-service layer.
- The definition of "having a subscription" remains vague and the subscription mechanism is not clearly described.
- The impact the create service element and the delete service element have on the OM-service, under which circumstances these services can be used, and the mechanism used by the create service element and the delete service element all are still unclear.

In order to find out about these obscurities, conversations with AVV are necessary. Furthermore, the dynamic behaviour of the OM-service and the OM-protocol are not validated by AVV, and may reveal (other than the in this Section mentioned) unwanted behaviour that is incorporated in the specifications.

6.5.2 Research on the Tools

Although the toolbox Eucalyptus offers quite a range of functionalities, there are still several items that deserve some extra attention. Three items, felt to be worth investigating are listed below:

- Extending the ADT-compiling algorithm with respect to the omission found, discussed on page 16.
- Performance analysis of the toolbox.
- Guidelines on the use of the compositional approach in generating LTSs.

Concerning the omission found on page 16: curing this omission is not felt to present much problems.

With respect to the performance analysis of the toolbox, it must be noted that it could be worthwhile to have some guidelines describing how a Lotos-description could best be written to reach faster results. This applies to both the static part, specified in act-one, as to the dynamic part.

With respect to the last item, it must be mentioned that the compositional generation of the LTS for the OM-protocol could have been done in more than one way. To illustrate: it was found out that restricting the RR-service model to a maximum of four messages that can be in transit at any point in time was too restrictive and introduced a deadlock. To cure this, one extra instance of process `communication` was added to the RR-service. Instead of generating the LTS for this new RR-service, the following construction was used to describe the new OM-protocol:

```
BEHAVIOUR (rr4 ||| rr1) || (APE ||| MPE)
```

In the above described construction **rr4** is a reference to the file *rr4.aut* which contains the LTS for the RR-service specification restricted to a maximum of four messages that can be in transit at any point in time, and **rr1** is a reference to the file *rr1.aut*, which contains the LTS for the RR-service specification restricted to at most one message that can be in transit at any point in time. **APE** and **MPE** refer to the files *APE.aut* and *MPE.aut*, which contain the LTSs for the Agent OM-protocol entity and the Manager OM-protocol entity. Generating the LTS for the new RR-service was thus avoided. Instead of the above described construction, however, the following construction could have easily been used as well:

```
BEHAVIOUR (rr1 ||| rr1 ||| rr1 ||| rr1 ||| rr1) || (APE ||| MPE)
```

It is interesting to investigate, which of these two constructions is more time and/or memory efficient in generating the corresponding LTS, and more general, if the choice between the two constructions can be automated. The ultimate challenge is to automate the compositional approach itself.

Another issue that can be investigated is the relation between an EFSM generated by SMILE and an LTS, generated by CÆSAR. It is interesting to investigate if, or under which circumstances, the restrictions imposed on the LOTOS-operators by CÆSAR can be overcome by transforming a specification into an EFSM first, and only then translate the EFSM into an LTS.

6.6 Recommendations for Similar Excercises

The approach followed in this thesis (i.e. first describing the specifications found in the Appendices, and only then modelling these specifications) did lead to some satisfactory results (e.g. absence of deadlock in the OM-protocol model and the OM-service could be proven).
Another approach is to first describe the models, validate these models and then gradually add more functionality to these models, ultimately giving a full specification.
Both approaches have their advantages and disadvantages.

The advantage of the approach followed in this thesis is that the models could be modelled after an already formal specification, thus easing the difficult task of determining what areas are critical and which are not.
A disadvantage, experienced during the traineeship is that it is very difficult to determine how many simplifications have to be made to the original specifications in order to obtain a model that can be validated by tools.

The other approach has as a disadvantage that initially it is very hard to distinguish the "vital" parts of the system under specification. As such, it is often hard to write a first model. Furthermore, once a model is described, it is hard to leave the already established framework behind if it is not suitable to add a new functionality to it.
An obvious advantage is that tool assistance is offered from the very beginning. The turning point of which functionalities incorporated in the models can still be validated by the tools and which cannot is easier to distinguish. Another advantage is that if the specification formalism is not well understood, this approach offers the possibility to gradually better understand the formalism.

It is hard to say which of these two approaches is best. Probably a mixture of both approaches yields the best (and fastest) results.

Appendix A

The OM-Service Specification

The requirements that can be found (although in no case, explicitly formulated) in [AVV97], or have been assumed, are the following :

1. A number of concurrent Associations are to be provided by the OM Service Provider — [AVV97].
2. An Association can be identified by its Manager and its Agent (thus the tuple (Manager,Agent) uniquely identifies the association) — This has been assumed —as stated before, [AVV97] does not acknowledge the roles of Managers and Agents —.
3. Once an association has started, it can only be terminated by a disable request. This disable request can be issued both by the Manager and the Agent and always succeeds — [AVV97].
4. Furthermore, a service provided termination is allowed — assumed.
5. Associations can only be set up between existing, different Managed Objects — assumed.
6. Both a disable indication and a disable confirmation denote the definite end of the corresponding association. — [AVV97]
7. A Managed Object can always be deleted when it is not one of the association members engaging in this operation — assumed.
8. A Managed Object can only be created when it does not already exist — assumed.
9. The Objects that are addressed, are identified by a dynamic set of object-ids, which, however, is not the task of the OM Service Provider to maintain — assumed.
10. Objects involved in an association cannot cease to exist spontaneously, but must first disable *all* associations they are involved in — assumed.

```
SPECIFICATION OM_Service [OSAP] : NOEXIT

LIBRARY
  boolean
  naturalnumber
ENDLIB

TYPE OM_Address
IS Boolean
SORTS OA
OPNS
  offset           : -> OA
  neighbour        : OA -> OA
  _eq_,_lt_        : OA, OA -> Bool
EQNS
FORALL
  x,y : OA
OFSORT
  Bool
  offset eq offset                  = true;
  offset eq neighbour(x)            = false;
  neighbour(x) eq offset            = false;
  neighbour(x) eq neighbour(y)      = x eq y;
  offset lt offset                  = false;
  offset lt neighbour(x)            = true;
  neighbour(x) lt offset            = false;
  neighbour(x) lt neighbour(y)      = x lt y;
```

```
ENDTYPE(*OM_Address*)


TYPE ID_Number
IS Boolean
SORTS ID
OPNS
  0                 : -> ID
  succ              : ID -> ID
  _eq_,_lt_         : ID, ID -> Bool
EQNS
FORALL
  x,y : ID
OFSORT Bool
  0 eq 0                             = true;
  0 eq succ(x)                       = false;
  succ(x) eq 0                       = false;
  succ(x) eq succ(y)                 = x eq y;

ENDTYPE(*ID_Number*)

TYPE AssociationSet
IS set ACTUALIZEDBY AId USING
SORTNAMES
  AId FOR Element
  AIdSet FOR Set
  Bool  FOR FBool
OPNNAMES
  empty FOR {}
ENDTYPE(*AssociationSet*)


TYPE AId
IS OM_Address, ID_Number, Boolean
SORTS
  AId
OPNS
  AId           : OA, OA -> AId
  Initiator     : AId -> OA
  Responder     : AId -> OA
  _eq_,_ne_,_lt_: AId, AId -> Bool
EQNS
FORALL
  a1,a2 : OA, as1,as2 : AId
OFSORT OA
  Initiator(AId(a1,a2))         = a1;
  Responder(AId(a1,a2))         = a2;
OFSORT Bool
  as1 eq as2                    = (Initiator(as1) eq Initiator(as2)) and
                                  (Responder(as1) eq Responder(as2));
  as1 ne as2                    = not(as1 eq as2);
  as1 lt as2                    = (Initiator(as1) lt Initiator(as2)) or
                                  ((Initiator(as1) eq Initiator(as2))
                                  and (Responder(as1) lt Responder(as2)));

ENDTYPE(*AId*)

TYPE OM_Service_Primitives
IS OM_Address, ID_Number, AId
SORTS
  OSP
OPNS
  getreq            : AID, ID -> OSP
  csetreq           : AID, ID -> OSP
  usetreq           : AID, ID -> OSP
  actionreq         : AID, ID -> OSP
  enablereq         : AID, ID -> OSP
  disablereq        : AID, ID -> OSP
  reportreq         : AID, ID -> OSP
  notificationreq   : AID, ID -> OSP
  createreq         : AID, ID, OA -> OSP
  deletereq         : AID, ID, OA -> OSP
  getind            : AID, ID -> OSP
  csetind           : AID, ID -> OSP
  usetind           : AID, ID -> OSP
  actionind         : AID, ID -> OSP
  enableind         : AID, ID -> OSP
  disableind        : AID, ID -> OSP
  reportind         : AID, ID -> OSP
  notificationind   : AID, ID -> OSP
  createind         : AID, ID, OA -> OSP
  deleteind         : AID, ID, OA -> OSP
  getres            : AID, ID -> OSP
  csetres           : AID, ID -> OSP
  actionres         : AID, ID -> OSP
  createres         : AID, ID, OA -> OSP
  deleteres         : AID, ID, OA -> OSP
  getconf           : AID, ID -> OSP
  csetconf          : AID, ID -> OSP
  actionconf        : AID, ID -> OSP
  enableconf        : AID, ID, Bool -> OSP
  disableconf       : AID, ID -> OSP
  createconf        : AID, ID, OA -> OSP
  deleteconf        : AID, ID, OA -> OSP
ENDTYPE(* OM_Service_Primitives*)


TYPE Doublet
IS Boolean, NaturalNumber
SORTS
  Tuplet
OPNS
  One, Two              : -> Tuplet
  _eq_, _ne_, _lt_      : Tuplet, Tuplet -> Bool
  h                     : Tuplet -> Nat
EQNS
FORALL x,y : Tuplet
OFSORT Nat
  h(One) = 0;
  h(Two) = succ(h(one));
OFSORT Bool
  x eq y = h(x) eq h(y);
  x ne y = h(x) ne h(y);
  x lt y = h(x) lt h(y);
ENDTYPE (*Doublet*)


TYPE EightTuplet
IS Doublet
OPNS
  Three, Four, Five, Six, Seven, Eight  :-> Tuplet
EQNS
OFSORT Nat
  h(Three)      = succ(h(Two));
  h(Four)       = succ(h(Three));
  h(Five)       = succ(h(Four));
  h(Six)        = succ(h(Five));
  h(Seven)      = succ(h(Six));
  h(Eight)      = succ(h(Seven));
ENDTYPE (*EightTuplet*)


TYPE SixteenTuplet
IS EightTuplet
OPNS
  Nine,Ten,Eleven,Twelve,Thirteen,Fourteen,
  Fifteen,Sixteen                               :-> Tuplet
EQNS
OFSORT Nat
  h(Nine)       = succ(h(Eight));
  h(Ten)        = succ(h(Nine));
  h(Eleven)     = succ(h(Ten));
  h(Twelve)     = succ(h(Eleven));
  h(Thirteen)   = succ(h(Twelve));
  h(Fourteen)   = succ(h(Thirteen));
  h(Fifteen)    = succ(h(Fourteen));
  h(Sixteen)    = succ(h(Fifteen))
ENDTYPE (*SixteenTuplet*)


TYPE ThirtytwoTuplet
IS SixteenTuplet
OPNS
  Seventeen, Eighteen, Nineteen, Twenty, Twone,
  Twtwo, Twthree, Twfour, Twfive, Twsix, Twseven,
  Tweight, Twnine, Thirty, Thone, Thtwo            :-> Tuplet
EQNS
OFSORT Nat
  h(Seventeen)  = succ(h(Sixteen));
  h(Eighteen)   = succ(h(Seventeen));
  h(Nineteen)    = succ(h(Eighteen));
  h(Twenty)      = succ(h(Nineteen));
  h(Twone)       = succ(h(Twenty));
  h(Twtwo)       = succ(h(Twone));
  h(TwThree)     = succ(h(TwTwo));
  h(TwFour)      = succ(h(TwThree));
  h(TwFive)      = succ(h(TwFour));
  h(TwSix)       = succ(h(TwFive));
  h(TwSeven)     = succ(h(TwSix));
  h(TwEight)     = succ(h(TwSeven));
  h(TwNine)      = succ(h(TwEight));
  h(Thirty)      = succ(h(Twnine));
  h(Thone)       = succ(h(Thirty));
  h(Thtwo)       = succ(h(Thone));
ENDTYPE (*ThirtytwoTuplet*)


TYPE OSPClass IS ThirtytwoTuplet RENAMEDBY
SORTNAMES
  OSPClass FOR Tuplet
OPNNAMES
    GetRequest                  FOR one
    USetRequest                 FOR two
    CSetRequest                 FOR three
    ActionRequest               FOR four
    EnableRequest               FOR five
    DisableRequest              FOR six
    ReportRequest               FOR seven
    NotificationRequest         FOR eight
    CreateRequest               FOR nine
    DeleteRequest               FOR ten
    GetIndication               FOR eleven
    USetIndication              FOR twelve
    CSetIndication              FOR thirteen
    ActionIndication            FOR fourteen
    EnableIndication            FOR fifteen
    DisableIndication           FOR sixteen
    ReportIndication            FOR seventeen
    NotificationIndication      FOR eighteen
    CreateIndication            FOR nineteen
    DeleteIndication            FOR twenty
    GetResponse                 FOR twone
    CSetResponse                FOR twtwo
    ActionResponse              FOR twthree
    CreateResponse              FOR twfour
    DeleteResponse              FOR twfive
    GetConfirmation             FOR twsix
    CSetConfirmation            FOR twseven
    ActionConfirmation          FOR tweight
    EnableConfirmation          FOR twnine
    DisableConfirmation         FOR thirty
    CreateConfirmation          FOR thone
    DeleteConfirmation          FOR thtwo

ENDTYPE(*OSP_Class*)

TYPE OSP_Classifier
IS OSPClass, OM_Service_Primitives, AId
OPNS
  Map               : OSP -> OSPClass
  IsGetReq          : OSP -> Bool
  IsCsetReq         : OSP -> Bool
  IsUsetReq         : OSP -> Bool
  IsActionReq       : OSP -> Bool
  IsEnableReq       : OSP -> Bool
  IsDisableReq      : OSP -> Bool
  IsReportReq       : OSP -> Bool
  IsNotificationReq : OSP -> Bool
  IsCreateReq       : OSP -> Bool
  IsDeleteReq       : OSP -> Bool
  IsGetInd          : OSP -> Bool
  IsCsetInd         : OSP -> Bool
  IsUsetInd         : OSP -> Bool
  IsActionInd       : OSP -> Bool
  IsEnableInd       : OSP -> Bool
  IsDisableInd      : OSP -> Bool
  IsReportInd       : OSP -> Bool
  IsNotificationInd : OSP -> Bool
  IsCreateInd       : OSP -> Bool
  IsDeleteInd       : OSP -> Bool
  IsGetRes          : OSP -> Bool
  IsCsetRes         : OSP -> Bool
  IsActionRes       : OSP -> Bool
  IsCreateRes       : OSP -> Bool
  IsDeleteRes       : OSP -> Bool
  IsGetConf         : OSP -> Bool
  IsCsetConf        : OSP -> Bool
  IsActionConf      : OSP -> Bool
  IsEnableConf      : OSP -> Bool
  IsDisableConf     : OSP -> Bool
  IsCreateConf      : OSP -> Bool
  IsDeleteConf      : OSP -> Bool
EQNS
FORALL
  a : OA, as1,as : AId, j : ID, prim : OSP, b : Bool
OFSORT OSPClass
  Map(GetReq(as,j))              = GetRequest;
  Map(USetReq(as,j))             = USetRequest;
  Map(CSetReq(as,j))             = CSetRequest;
  Map(ActionReq(as,j))           = ActionRequest;
  Map(EnableReq(as,j))           = EnableRequest;
  Map(DisableReq(as,j))          = DisableRequest;
  Map(ReportReq(as,j))           = ReportRequest;
  Map(NotificationReq(as,j))     = NotificationRequest;
  Map(CreateReq(as,j,a))         = CreateRequest;
  Map(DeleteReq(as,j,a))         = DeleteRequest;
  Map(GetInd(as,j))              = GetIndication;
  Map(USetInd(as,j))             = USetIndication;
  Map(CSetInd(as,j))             = CSetIndication;
  Map(ActionInd(as,j))           = ActionIndication;
  Map(EnableInd(as,j))           = EnableIndication;
  Map(DisableInd(as,j))          = DisableIndication;
  Map(ReportInd(as,j))           = ReportIndication;
  Map(NotificationInd(as,j))     = NotificationIndication;
  Map(CreateInd(as,j,a))         = CreateIndication;
  Map(DeleteInd(as,j,a))         = DeleteIndication;
  Map(GetRes(as,j))              = GetResponse;
  Map(CSetRes(as,j))             = CSetResponse;
  Map(ActionRes(as,j))           = ActionResponse;
  Map(CreateRes(as,j,a))         = CreateResponse;
  Map(DeleteRes(as,j,a))         = DeleteResponse;
  Map(GetConf(as,j))             = GetConfirmation;
  Map(CSetConf(as,j))            = CSetConfirmation;
  Map(ActionConf(as,j))          = ActionConfirmation;
  Map(EnableConf(as,j,b))        = EnableConfirmation;
  Map(DisableConf(as,j))         = DisableConfirmation;
  Map(CreateConf(as,j,a))        = CreateConfirmation;
  Map(DeleteConf(as,j,a))        = DeleteConfirmation;
OFSORT Bool
  IsGetReq(prim)                   = map(prim) eq getrequest;
  IsCsetReq(prim)                  = map(prim) eq CSetRequest;
  IsUsetReq(prim)                  = map(prim) eq USetRequest;
  IsActionReq(prim)                = map(prim) eq ActionRequest;
  IsEnableReq(prim)                = map(prim) eq EnableRequest;
  IsDisableReq(prim)               = map(prim) eq DisableRequest;
  IsReportReq(prim)                = map(prim) eq ReportRequest;
  IsNotificationReq(prim)          = map(prim) eq NotificationRequest;
  IsCreateReq(prim)                = map(prim) eq CreateRequest;
  IsDeleteReq(prim)                = map(prim) eq DeleteRequest;
  IsGetInd(prim)                   = map(prim) eq GetIndication;
  IsCsetInd(prim)                  = map(prim) eq USetIndication;
  IsUsetInd(prim)                  = map(prim) eq CSetIndication;
  IsActionInd(prim)                = map(prim) eq ActionIndication;
  IsEnableInd(prim)                = map(prim) eq EnableIndication;
  IsDisableInd(prim)               = map(prim) eq DisableIndication;
  IsReportInd(prim)                = map(prim) eq ReportIndication;
  IsNotificationInd(prim)          = map(prim) eq NotificationIndication;
  IsCreateInd(prim)                = map(prim) eq CreateIndication;
  IsDeleteInd(prim)                = map(prim) eq DeleteIndication;
  IsGetRes(prim)                   = map(prim) eq GetResponse;
  IsCsetRes(prim)                  = map(prim) eq CSetResponse;
  IsActionRes(prim)                = map(prim) eq ActionResponse;
  IsCreateRes(prim)                = map(prim) eq CreateResponse;
  IsDeleteRes(prim)                = map(prim) eq DeleteResponse;
  IsGetConf(prim)                  = map(prim) eq GetConfirmation;
  IsCsetConf(prim)                 = map(prim) eq CSetConfirmation;
  IsActionConf(prim)               = map(prim) eq ActionConfirmation;
  IsEnableConf(prim)               = map(prim) eq EnableConfirmation;
  IsDisableConf(prim)              = map(prim) eq DisableConfirmation;
  IsCreateConf(prim)               = map(prim) eq CreateConfirmation;
  IsDeleteConf(prim)                = map(prim) eq DeleteConfirmation;
ENDTYPE(*OSP_Classifier*)


TYPE OSP_Attributes
IS OSP_Classifier, AId, Boolean
OPNS
  GetId                 : OSP -> ID
  Subj                  : OSP -> OA
  GetAId                : OSP -> AId
  GetSuccess            : OSP -> Bool
EQNS
FORALL
  a : OA, j : ID, as : AId, b : Bool
OFSORT Bool
  GetSuccess(EnableConf(as,j,b)) = b;
OFSORT ID
  GetId(GetReq(as,j))              = j;
  GetId(USetReq(as,j))             = j;
  GetId(CSetReq(as,j))             = j;
  GetId(ActionReq(as,j))           = j;
  GetId(EnableReq(as,j))           = j;
  GetId(DisableReq(as,j))          = j;
  GetId(ReportReq(as,j))           = j;
  GetId(NotificationReq(as,j))     = j;
  GetId(CreateReq(as,j,a))         = j;
  GetId(DeleteReq(as,j,a))         = j;
  GetId(GetInd(as,j))              = j;
  GetId(USetInd(as,j))             = j;
  GetId(CSetInd(as,j))             = j;
  GetId(ActionInd(as,j))           = j;
  GetId(EnableInd(as,j))           = j;
  GetId(DisableInd(as,j))          = j;
  GetId(ReportInd(as,j))           = j;
  GetId(NotificationInd(as,j))     = j;
  GetId(CreateInd(as,j,a))         = j;
  GetId(DeleteInd(as,j,a))         = j;
  GetId(GetRes(as,j))              = j;
  GetId(CSetRes(as,j))             = j;
  GetId(ActionRes(as,j))           = j;
  GetId(CreateRes(as,j,a))         = j;
  GetId(DeleteRes(as,j,a))         = j;
  GetId(GetConf(as,j))             = j;
  GetId(CSetConf(as,j))            = j;
  GetId(ActionConf(as,j))          = j;
  GetId(EnableConf(as,j,b))        = j;
  GetId(DisableConf(as,j))         = j;
  GetId(CreateConf(as,j,a))        = j;
  GetId(DeleteConf(as,j,a))        = j;
OFSORT OA
  Subj(CreateReq(as,j,a))         = a;
  Subj(DeleteReq(as,j,a))         = a;
  Subj(CreateInd(as,j,a))         = a;
  Subj(DeleteInd(as,j,a))         = a;
  Subj(CreateRes(as,j,a))         = a;
  Subj(DeleteRes(as,j,a))         = a;
  Subj(CreateConf(as,j,a))        = a;
  Subj(DeleteConf(as,j,a))        = a;
OFSORT AId
  GetAId(GetReq(as,j))              = as;
  GetAID(USetReq(as,j))             = as;
  GetAID(CSetReq(as,j))             = as;
  GetAID(ActionReq(as,j))           = as;
  GetAID(EnableReq(as,j))           = as;
  GetAID(DisableReq(as,j))          = as;
  GetAID(ReportReq(as,j))           = as;
  GetAID(NotificationReq(as,j))     = as;
  GetAID(CreateReq(as,j,a))         = as;
  GetAID(DeleteReq(as,j,a))         = as;
  GetAID(GetInd(as,j))              = as;
  GetAID(USetInd(as,j))             = as;
  GetAID(CSetInd(as,j))             = as;
  GetAID(ActionInd(as,j))           = as;
  GetAID(EnableInd(as,j))           = as;
  GetAID(DisableInd(as,j))          = as;
  GetAID(ReportInd(as,j))           = as;
  GetAID(NotificationInd(as,j))     = as;
  GetAID(CreateInd(as,j,a))         = as;
  GetAID(DeleteInd(as,j,a))         = as;
  GetAID(GetRes(as,j))              = as;
  GetAID(CSetRes(as,j))             = as;
  GetAID(ActionRes(as,j))           = as;
  GetAID(CreateRes(as,j,a))         = as;
  GetAID(DeleteRes(as,j,a))         = as;
  GetAID(GetConf(as,j))             = as;
  GetAID(CSetConf(as,j))            = as;
  GetAID(ActionConf(as,j))          = as;
  GetAID(EnableConf(as,j,b))        = as;
  GetAID(DisableConf(as,j))         = as;
  GetAID(CreateConf(as,j,a))        = as;
  GetAID(DeleteConf(as,j,a))        = as;

ENDTYPE(*OSP_Attributes*)

TYPE OSP_ServiceTYPE
IS OSP_Classifier

OPNS
  IsReq                 : OSP -> Bool
  IsInd                 : OSP -> Bool
  IsRes                 : OSP -> Bool
  IsConf                : OSP -> Bool

EQNS
FORALL
  prim : OSP
OFSORT bool

  IsReq(prim)           = IsGetReq(prim) or IsCSetReq(prim) or
                          IsUSetReq(prim) or IsActionReq(prim) or
                          IsEnableReq(prim) or IsDisableReq(prim) or
                          IsCreateReq(prim) or IsDeleteReq(prim) or
                          IsReportReq(prim) or IsNotificationReq(prim);
  IsInd(prim)           = IsGetInd(prim) or IsCSetInd(prim) or
                          IsUSetInd(prim) or IsActionInd(prim) or
                          IsEnableInd(prim) or IsDisableInd(prim) or
                          IsCreateInd(prim) or IsDeleteInd(prim) or
                          IsReportInd(prim) or IsNotificationInd(prim);
  IsRes(prim)           = IsGetRes(prim) or IsCSetRes(prim) or
                          IsActionRes(prim) or
                          IsCreateRes(prim) or IsDeleteRes(prim);
  IsConf(prim)          = IsGetConf(prim) or IsCSetConf(prim) or
                          IsActionConf(prim) or
                          IsEnableConf(prim) or IsDisableConf(prim) or
                          IsCreateConf(prim) or IsDeleteConf(prim);

ENDTYPE (* OSP_ServiceTYPE *)

TYPE OSP_Elements is OSP_Classifier, Boolean
OPNS
  IsGet             : OSP -> Bool
  IsCSet            : OSP -> Bool
  IsUSet            : OSP -> Bool
  IsAction          : OSP -> Bool
  IsEnable          : OSP -> Bool
  IsDisable         : OSP -> Bool
  IsReport          : OSP -> Bool
  IsNotification    : OSP -> Bool
  IsCreate          : OSP -> Bool
  IsDelete          : OSP -> Bool
EQNS
FORALL
  prim : OSP
OFSORT bool
  IsGet(prim)         = IsGetReq(prim) or IsGetInd(prim) or
                        IsGetRes(prim) or IsGetConf(prim);
  IsCSet(prim)        = IsCSetReq(prim) or IsCSetInd(prim) or
                        IsCSetRes(prim) or IsCSetConf(prim);
  IsAction(prim)      = IsActionReq(prim) or IsActionInd(prim) or
                        IsActionRes(prim) or IsActionConf(prim);
  IsUSet(prim)        = IsUSetReq(prim) or IsUSetInd(prim);
  IsEnable(prim)      = IsEnableReq(prim) or IsEnableInd(prim) or
                        IsEnableConf(prim);
  IsDisable(prim)     = IsDisableReq(prim) or IsDisableInd(prim) or
                        IsDisableConf(prim);
  IsReport(prim)      = IsReportReq(prim) or IsReportInd(prim);
  IsNotification(prim)= IsNotificationReq(prim) or IsNotificationInd(prim);
  IsCreate(prim)      = IsCreateReq(prim) or IsCreateInd(prim) or
                        IsCreateRes(prim) or IsCreateConf(prim);
  IsDelete(prim)      = IsDeleteReq(prim) or IsDeleteInd(prim) or
```

```
                        IsDeleteRes(prim) or IsDeleteConf(prim);
ENDTYPE(*OSP_Elements*)


TYPE OSP_Type
IS
  Boolean, OSP_Elements

OPNS

  IsUnConf          : OSP -> Bool
  IsUserConf        : OSP -> Bool
  IsProviderConf    : OSP -> Bool

EQNS
FORALL prim : OSP

OFSORT Bool
  IsUnConf(prim)        = IsUSet(prim) or IsReport(prim) or
                          IsNotification(prim);
  IsUserConf(prim)      = IsGet(prim) or IsCSet(prim) or
                          IsAction(prim) or IsCreate(prim) or
                          IsDelete(prim);
  IsProviderConf(prim)  = IsEnable(prim) or IsDisable(prim);
ENDTYPE (* OSP_Type *)

TYPE OSP_Match
IS OSP_Elements, Boolean
OPNS
  _matches_   : OSP, OSP -> Bool
EQNS
FORALL p,q : OSP
OFSORT Bool
  p matches q    = (Isdisable(p) and Isdisable(q)) or
                   (IsEnable(p) and IsEnable(q)) or
                   (IsGet(p) and IsGet(q)) or
                   (IsCSet(p) and IsCSet(q)) or
                   (IsUSet(p) and IsUSet(q)) or
                   (IsAction(p) and IsAction(q)) or
                   (IsCreate(p) and IsCreate(q)) or
                   (IsReport(p) and IsReport(q)) or
                   (IsDelete(p) and IsDelete(q)) or
                   (IsNotification(p) and IsNotification(q));
ENDTYPE (* OSP_Match *)

TYPE OSP_Primitives_Relations
IS OSP_Classifier, Boolean, OSP_Attributes, ID_Number,OSP_Servicetype,OSP_Match
OPNS
  _ConfForReq_      : OSP, OSP -> Bool
  _ConfForRes_      : OSP, OSP -> Bool
  _IndForReq_       : OSP, OSP -> Bool
  _ResForInd_       : OSP, OSP -> Bool
EQNS
FORALL prim1,prim2 : OSP
OFSORT Bool
  prim1 ConfForReq prim2      = IsConf(prim1) and IsReq(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);
  prim1 ConfForRes prim2      = IsConf(prim1) and IsRes(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);
  prim1 IndForReq prim2       = IsInd(prim1) and IsReq(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);
  prim1 ResForInd prim2       = IsRes(prim1) and IsInd(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);

ENDTYPE(*OSP_Primitives_Relations*)


BEHAVIOUR

  associations[OSAP](insert(aid(offset,neighbour(offset)),
                  (insert(aid(neighbour(offset),offset), empty))))

WHERE


PROCESS associations[OSAP](A : aidset) : NOEXIT :=
  CHOICE aa : aid [] [(aa isin A) and not(initiator(aa) eq responder(aa))] ->
   (i; associations[OSAP](remove(aa,A) )
   |||
    association[OSAP](aa)
   )

ENDPROC(*associations*)

PROCESS association[OSAP](aa : aid) : NOEXIT :=

  ( manager[OSAP](aa) ||| agent[OSAP](aa) )
  ||
    sync[OSAP](aa)

ENDPROC (*association*)

PROCESS manager[OSAP](aa : aid) : NOEXIT :=

  minitiate[OSAP](aa)
  >>
    mcommunicate[OSAP](aa)
  ||
    initiaterelease[OSAP](initiator(aa))
  [>
    (forcerelease[OSAP](initiator(aa)) >> manager[OSAP](aa) )

ENDPROC (*manager*)


PROCESS agent[OSAP](aa : aid) : NOEXIT :=

  ainitiate[OSAP](aa)
  >>
    acommunicate[OSAP](aa)
  ||
    initiaterelease[OSAP](responder(aa))
  [>
    (forcerelease[OSAP](responder(aa)) >> agent[OSAP](aa) )

ENDPROC (*agent*)

PROCESS minitiate[OSAP](aa : aid) : EXIT :=

  OSAP
    !initiator(aa)
    ?p1 : OSP[IsEnableReq(p1) and (aa eq GetAID(p1))];
  OSAP
    !initiator(aa)
    ?p2 : OSP[p2 ConfForReq p1];
  ([not(GetSuccess(p2))] -> minitiate[OSAP](aa)
  []
   [GetSuccess(p2)] -> EXIT)

ENDPROC (*minitiate*)


PROCESS ainitiate[OSAP](aa : aid) : EXIT :=

     OSAP
       !responder(aa)
       ?p1 : OSP[IsEnableInd(p1) and (aa eq GetAID(p1))];
     EXIT

ENDPROC (*ainitiate*)

PROCESS mcommunicate[OSAP](aa : aid) : NOEXIT :=

  mnsm[Osap](aa) ||| mosm[Osap](aa)

ENDPROC (* mcommunicate *)

PROCESS acommunicate[Osap](aa : aid) : NOEXIT :=

  mnsa[Osap](aa) ||| mosa[Osap](aa)

ENDPROC (* acommunicate *)

PROCESS mosm[OSAP](aa : aid) : NOEXIT :=
  mpconfirmed[OSAP](initiator(aa))
|||
  munconfirmed[OSAP](initiator(aa))
|||
  (  muserconfirmed[OSAP](initiator(aa))
  || validoperations[OSAP](initiator(aa)))
|||
  i;
  mosm[OSAP](aa)

ENDPROC (* mosm *)


PROCESS mosa[OSAP](aa : aid) : NOEXIT :=

  apconfirmed[OSAP](responder(aa))
|||
  aunconfirmed[OSAP](responder(aa))
|||
  auserconfirmed[OSAP](responder(aa))
|||
  i;
  mosa[OSAP](aa)

ENDPROC (* mosa *)

PROCESS mpconfirmed[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsDisableReq(p1)];
  EXIT

ENDPROC (* mpconfirmed *)

PROCESS apconfirmed[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsDisableReq(p1)];
  EXIT

ENDPROC (* apconfirmed *)


PROCESS munconfirmed[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsUsetReq(p1)];
  EXIT

ENDPROC (* munconfirmed *)


PROCESS aunconfirmed[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsUsetInd(p1)];
  EXIT

ENDPROC (* aunconfirmed *)


PROCESS muserconfirmed[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsReq(p1) and IsUserConf(p1)];
  OSAP
    !x
    ?p2 : OSP[p2 ConfForReq p1];
  EXIT

ENDPROC (* muserconfirmed *)


PROCESS auserconfirmed[OSAP](x : oa) : EXIT :=
  OSAP
    !x
    ?p1 : OSP[IsInd(p1) and IsUserConf(p1)];
  OSAP
    !x
    ?p2 : OSP[p2 ResForInd p1];
  EXIT

ENDPROC (* auserconfirmed *)


PROCESS validoperations[OSAP](x : oa) : NOEXIT :=

  OSAP
    !x
    ?p1 : OSP[(IsDelete(p1) or IsCreate(p1)) implies
              not((subj(p1) eq initiator(getaid(p1))) or
                  (subj(p1) eq responder(getaid(p1))))];
  validoperations[OSAP](x)

ENDPROC

PROCESS mnsm[OSAP](aa : aid) : NOEXIT :=

  mreport[OSAP](initiator(aa))
|||
  mnotification[OSAP](initiator(aa))
|||
  i;
  mnsm[OSAP](aa)

ENDPROC (* mnsm *)


PROCESS mnsa[OSAP](aa : aid) : NOEXIT :=

  areport[OSAP](responder(aa))
|||
  anotification[OSAP](responder(aa))
|||
  i;
  mnsa[OSAP](aa)

ENDPROC (* mnsa *)


PROCESS mreport[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsReportInd(p1)];
  EXIT

ENDPROC (* mreport *)


PROCESS mnotification[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[Isnotification(p1)];
  EXIT

ENDPROC (* mnotification *)


PROCESS areport[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsReportReq(p1)];
  EXIT

ENDPROC (* areport *)


PROCESS anotification[OSAP](x : oa) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[Isnotification(p1)];
  EXIT

ENDPROC (* anotification *)

PROCESS forcerelease[OSAP](x : OA) : EXIT :=

  OSAP
    !x
    ?p1 : OSP[IsDisableConf(p1) or IsDisableInd(p1)];
  EXIT

ENDPROC (* forcerelease *)

PROCESS initiaterelease[OSAP](x : OA) : NOEXIT :=

  OSAP
    !x
    ?p1 : OSP;
  ([not(IsDisableReq(p1))] -> initiaterelease[OSAP](x)
  []
   [IsDisableReq(p1)] -> releasing[OSAP](x)
  )
WHERE

  PROCESS releasing[Osap](y : OA) : NOEXIT :=

    OSAP
      !y
      ?p1 : OSP[IsConf(p1) or IsDisableInd(p1)];
    releasing[Osap](y)

  ENDPROC (* releasing *)

ENDPROC (* initiaterelease *)

PROCESS sync[OSAP](aa : aid) : NOEXIT :=

   (pcsync[OSAP](aa)
  |||
    ucsync[OSAP](aa)
  |||
    unsync[OSAP](aa) )
  ||
    identification[OSAP](aa)

WHERE

  PROCESS identification[OSAP](aa : aid) : NOEXIT :=

    OSAP
      !initiator(aa)
      ?p1 : OSP[aa eq getaid(p1)];
    identification[OSAP](aa)
  []
    OSAP
      !responder(aa)
      ?p1 : OSP[aa eq getaid(p1)];
    identification[OSAP](aa)

  ENDPROC (* identification *)

  PROCESS ucsync[OSAP](aa : aid) : NOEXIT :=

      OSAP
        !initiator(aa)
        ?p1 : OSP[IsReq(p1) and IsUserConf(p1)];
      OSAP
        !responder(aa)
        ?p2 : OSP[p2 IndForReq p1];
      EXIT
    |||
      OSAP
        !responder(aa)
        ?p1 : OSP[IsRes(p1) and IsUserConf(p1)];
      OSAP
        !initiator(aa)
        ?p2 : OSP[p2 ConfForRes p1];
      EXIT
    |||
    i;
    ucsync[OSAP](aa)

ENDPROC (* ucsync *)


PROCESS unsync[OSAP](aa : aid) : NOEXIT :=

    OSAP
      ?n  : OA
      ?p1 : OSP[((n eq initiator(aa)) or (n eq responder(aa))) and IsReq(p1) and IsUnConf(p1)];
    OSAP
      ?n  : OA
      ?p2 : OSP[((n eq initiator(aa)) or (n eq responder(aa))) and (p2 IndForReq p1)];
    EXIT
  |||
    i;
    unsync[OSAP](aa)

ENDPROC (* uncsync *)

PROCESS pcsync[OSAP](aa : aid) : NOEXIT :=

    sinitiate[OSAP](aa)
  >>
    ( srelease[OSAP](initiator(aa), responder(aa) )
    []
      srelease[OSAP](responder(aa), initiator(aa) )
    )
  >>
    pcsync[OSAP](aa)

WHERE


  PROCESS sinitiate[OSAP](aa : aid) : EXIT :=

    OSAP
      !initiator(aa)
      ?p1 : OSP[IsEnableReq(p1)];
    ( OSAP
        !initiator(aa)
        ?p2 : OSP[IsEnableConf(p2) and not(GetSuccess(p2))];
      sinitiate[OSAP](aa)
    []
      OSAP
        !responder(aa)
        ?p2 : OSP[p2 IndForReq p1];
      OSAP
        !initiator(aa)
        ?p2 : OSP[IsEnableConf(p2) and GetSuccess(p2)];
     EXIT
    )

  ENDPROC (* sinitiate *)


  PROCESS srelease[OSAP](x,y : OA) : EXIT :=

    OSAP
      !x
      ?p1 : OSP[IsDisableReq(p1)];
    (
      OSAP
        !y
        ?p2 : OSP[IsDisableReq(p2)];
      (
        (scenario1[OSAP](x,y,p1) >> EXIT)
      []
        (scenario1[OSAP](y,x,p2) >> EXIT)
      []
        (scenario3[OSAP](x,y,p1,p2) >> EXIT)
      []
        (scenario3[OSAP](y,x,p2,p1) >> EXIT)
      )
    []
      (scenario1[OSAP](x,y,p1) >> EXIT)
    []
      (scenario2[OSAP](x,y) >> EXIT)
    []
    (scenario5[OSAP](x,y,p1) >> EXIT)
  )
[]
  (scenario4[OSAP](x,y) >> EXIT)

WHERE

  PROCESS scenario1[OSAP](x,y : oa, p : osp) : EXIT :=

    i;(* decide that process x should terminate *)
    OSAP
      !x
      ?p1 : OSP[IsDisableInd(p1)];
  (
    OSAP
      !y
      ?p1 : OSP[p1 IndForReq p];
    EXIT
  []
    i;(* decide that process y should terminate *)
    OSAP
      !y
      ?p1 : OSP[IsDisableInd(p1)];
    EXIT
  )

  ENDPROC (* scenario1 *)

  PROCESS scenario2[OSAP](x,y : oa) : EXIT :=

    i;(* decide that process x and y should terminate *)
    OSAP
      !y
      ?p1 : OSP[IsDisableInd(p1)];
    OSAP
      !x
      ?p1 : OSP[IsDisableInd(p1)];
    EXIT

  ENDPROC (* scenario2 *)

  PROCESS scenario3[OSAP](x,y : oa, p,q : osp) : EXIT :=

    OSAP
      !x
      ?p1 : OSP[p1 IndForReq q];
  (
    OSAP
      !y
      ?p1 : OSP[IsDisableConf(p1)];
    EXIT
  []
    OSAP
      !y
      ?p1 : OSP[p1 IndForReq p];
    EXIT
  []
    i;(* decide that process y should terminate *)
    OSAP
      !y
      ?p1 : OSP[IsDisableInd(p1)];
    EXIT
  )

  ENDPROC (* scenario3 *)

  PROCESS scenario4[OSAP](x,y : oa) : EXIT :=

    i;(* decide that process x should terminate *)
    OSAP
      !x
      ?p1 : OSP[IsDisableInd(p1)];
  (
    OSAP
      !y
      ?p1 : OSP[IsDisableReq(p1)];
    OSAP
      !x
      ?p1 : OSP[IsDisableInd(p1)];
    EXIT
```

```
        []
          i;(* decide that process y should terminate *)
          OSAP
            !y
            ?p1 : OSP[IsDisableInd(p1)];
          EXIT
        )

        ENDPROC (* scenario4 *)


      PROCESS scenario5[OSAP](x,y : oa,p : osp) : EXIT :=

        OSAP
          !y
          ?p1 : OSP[p1 IndForReq p];
      (
        OSAP
          !x
          ?p1 : OSP[IsDisableConf(p1)];
        EXIT
      []
        i;(* decide that process x should terminate *)
        OSAP
          !x
          ?p1 : OSP[IsDisableInd(p1)];
        EXIT
      )

      ENDPROC (* scenario5 *)

      ENDPROC (* srelease *)

    ENDPROC (* pcsync *)

  ENDPROC (*sync*)

ENDSPEC (*OM_Spec*)
```

Appendix B

The OM-protocol Specification

```
SPECIFICATION OM_Protocol[OSAP] : NOEXIT
LIBRARY
  set,NaturalNumber, boolean, element
ENDLIB


TYPE RR_Service_Primitives
IS Address_Identifier, ID_Number, dataunits
SORTS
  RSP
OPNS
  invokereq         : AI, AI, ID, OMPDU -> RSP
  invokeind         : AI, AI, ID, OMPDU -> RSP
  invokeconf        : AI, AI, ID, OMPDU, BOOL -> RSP
  resultreq         : AI, AI, ID, OMPDU -> RSP
  resultind         : AI, AI, ID, OMPDU -> RSP
  errorreq          : AI, AI, ID, OMPDU -> RSP
  errorind          : AI, AI, ID, OMPDU -> RSP
  eventreportreq    : AI, AI, ID, OMPDU -> RSP
  eventreportind    : AI, AI, ID, OMPDU -> RSP
  notificationreq   : AI, AI, ID, OMPDU -> RSP
  notificationind   : AI, AI, ID, OMPDU -> RSP
  rejectreq         : AI, AI, ID, OMPDU -> RSP
  rejectind         : AI, AI, ID, OMPDU -> RSP
ENDTYPE(*RSP_Primitives*)


TYPE RSPClass IS SixteenTuplet RENAMEDBY
SORTNAMES
  RSPClass FOR Tuplet
OPNNAMES
    InvokeRequest               FOR one
    ResultRequest               FOR two
    ErrorRequest                FOR three
    EventReportRequest          FOR four
    NotificationRequest         FOR five
    InvokeIndication            FOR six
    ResultIndication            FOR seven
    ErrorIndication             FOR eight
    EventReportIndication       FOR nine
    NotificationIndication      FOR ten
    InvokeConfirmation          FOR eleven
    RejectRequest               FOR twelve
    RejectIndication            FOR thirteen

ENDTYPE(*RSP_Class*)


TYPE RSP_Classifier
IS RSPClass, RR_Service_Primitives, DataUnits
OPNS
  Map                   : RSP -> RSPClass
  IsInvokeReq           : RSP -> Bool
  IsResultReq           : RSP -> Bool
  IsErrorReq            : RSP -> Bool
  IsEventReportReq      : RSP -> Bool
  IsNotificationReq     : RSP -> Bool
  IsInvokeInd           : RSP -> Bool
  IsResultInd           : RSP -> Bool
  IsErrorInd            : RSP -> Bool
  IsEventReportInd      : RSP -> Bool
  IsNotificationInd     : RSP -> Bool
  IsInvokeConf          : RSP -> Bool
  IsRejectReq           : RSP -> Bool
  IsRejectInd           : RSP -> Bool

EQNS
FORALL
   a1,a2 : AI, j : ID, prim : RSP, o : OMPDU, b : BOOL
OFSORT RSPClass
  Map(InvokeReq(a1,a2,j,o))            = InvokeRequest;
  Map(ResultReq(a1,a2,j,o))            = ResultRequest;
  Map(ErrorReq(a1,a2,j,o))             = ErrorRequest;
  Map(EventReportReq(a1,a2,j,o))       = EventReportRequest;
  Map(NotificationReq(a1,a2,j,o))      = NotificationRequest;
  Map(InvokeInd(a1,a2,j,o))            = InvokeIndication;
  Map(ResultInd(a1,a2,j,o))            = ResultIndication;
  Map(ErrorInd(a1,a2,j,o))             = ErrorIndication;
  Map(EventReportInd(a1,a2,j,o))       = EventReportIndication;
  Map(NotificationInd(a1,a2,j,o))      = NotificationIndication;
  Map(invokeconf(a1,a2,j,o,b))         = InvokeConfirmation;
  Map(RejectReq(a1,a2,j,o))            = RejectRequest;
  Map(RejectInd(a1,a2,j,o))            = RejectIndication;

OFSORT Bool
  IsInvokeReq(prim)                 = map(prim) eq Invokerequest;
  IsResultReq(prim)                 = map(prim) eq ResultRequest;
  IsErrorReq(prim)                  = map(prim) eq ErrorRequest;
  IsEventReportReq(prim)            = map(prim) eq EventReportRequest;
  IsNotificationReq(prim)           = map(prim) eq NotificationRequest;
  IsInvokeInd(prim)                 = map(prim) eq InvokeIndication;
  IsResultInd(prim)                 = map(prim) eq ResultIndication;
  IsErrorInd(prim)                  = map(prim) eq ErrorIndication;
  IsEventReportInd(prim)            = map(prim) eq EventReportIndication;
  IsNotificationInd(prim)           = map(prim) eq NotificationIndication;
  IsInvokeConf(prim)                = map(prim) eq InvokeConfirmation;
  IsRejectReq(prim)                 = map(prim) eq RejectRequest;
  IsRejectInd(prim)                 = map(prim) eq RejectIndication;

ENDTYPE(*RSP_Classifier*)


TYPE RSP_Attributes
IS RSP_Classifier, Boolean, DataUnits
OPNS
  GetId                 : RSP -> ID
  Inv                   : RSP -> AI
  Perf                  : RSP -> AI
  pdu                   : RSP -> OMPDU
  IsAck                 : RSP -> BOOL
  IsNack                : RSP -> BOOL
EQNS
FORALL
  a1,a2 : AI, j : ID, o : ompdu, b : bool
OFSORT ID
  GetId(InvokeReq(a1,a2,j,o))           = j;
  GetId(ResultReq(a1,a2,j,o))           = j;
  GetId(ErrorReq(a1,a2,j,o))            = j;
  GetId(EventReportReq(a1,a2,j,o))      = j;
  GetId(NotificationReq(a1,a2,j,o))     = j;
  GetId(InvokeInd(a1,a2,j,o))           = j;
  GetId(ResultInd(a1,a2,j,o))           = j;
  GetId(ErrorInd(a1,a2,j,o))            = j;
  GetId(EventReportInd(a1,a2,j,o))      = j;
  GetId(NotificationInd(a1,a2,j,o))     = j;
  GetId(InvokeConf(a1,a2,j,o,b))        = j;
  GetId(RejectReq(a1,a2,j,o))           = j;
  GetId(RejectInd(a1,a2,j,o))           = j;

OFSORT AI
  Inv(InvokeReq(a1,a2,j,o))             = a1;
  Inv(ResultReq(a1,a2,j,o))             = a1;
  Inv(ErrorReq(a1,a2,j,o))              = a1;
  Inv(EventReportReq(a1,a2,j,o))        = a1;
  Inv(NotificationReq(a1,a2,j,o))       = a1;
  Inv(InvokeInd(a1,a2,j,o))             = a1;
  Inv(ResultInd(a1,a2,j,o))             = a1;
  Inv(ErrorInd(a1,a2,j,o))              = a1;
  Inv(EventReportInd(a1,a2,j,o))        = a1;
  Inv(NotificationInd(a1,a2,j,o))       = a1;
  Inv(invokeconf(a1,a2,j,o,b))          = a1;
  Inv(RejectReq(a1,a2,j,o))             = a1;
  Inv(RejectInd(a1,a2,j,o))             = a1;

OFSORT AI
  Perf(InvokeReq(a1,a2,j,o))            = a2;
  Perf(ResultReq(a1,a2,j,o))            = a2;
  Perf(ErrorReq(a1,a2,j,o))             = a2;
  Perf(EventReportReq(a1,a2,j,o))       = a2;
  Perf(NotificationReq(a1,a2,j,o))      = a2;
  Perf(InvokeInd(a1,a2,j,o))            = a2;
  Perf(ResultInd(a1,a2,j,o))            = a2;
  Perf(ErrorInd(a1,a2,j,o))             = a2;
  Perf(EventReportInd(a1,a2,j,o))       = a2;
  Perf(NotificationInd(a1,a2,j,o))      = a2;
  Perf(invokeconf(a1,a2,j,o,b))         = a2;
  Perf(RejectReq(a1,a2,j,o))            = a2;
  Perf(RejectInd(a1,a2,j,o))            = a2;

OFSORT BOOL

  IsAck(invokeconf(a1,a2,j,o,b))        =  b;
  IsNack(invokeconf(a1,a2,j,o,b))       = not(b);

OFSORT OMPDU

  pdu(InvokeReq(a1,a2,j,o))             = o;
  pdu(ResultReq(a1,a2,j,o))             = o;
  pdu(ErrorReq(a1,a2,j,o))              = o;
  pdu(EventReportReq(a1,a2,j,o))        = o;
  pdu(NotificationReq(a1,a2,j,o))       = o;
  pdu(InvokeInd(a1,a2,j,o))             = o;
  pdu(ResultInd(a1,a2,j,o))             = o;
  pdu(ErrorInd(a1,a2,j,o))              = o;
  pdu(EventReportInd(a1,a2,j,o))        = o;
  pdu(NotificationInd(a1,a2,j,o))       = o;
  pdu(invokeconf(a1,a2,j,o,b))          = o;
  pdu(RejectReq(a1,a2,j,o))             = o;
  pdu(RejectInd(a1,a2,j,o))             = o;

ENDTYPE(*RSP_Attributes*)


TYPE RSP_ServiceType
IS RSP_Classifier, RSP_ServiceElements

OPNS
  IsReq                 : RSP -> Bool
  IsInd                 : RSP -> Bool
  IsConf                : RSP -> Bool

EQNS
FORALL
  prim : RSP
OFSORT bool

  IsReq(prim)           = IsInvokeReq(prim) or IsResultReq(prim) or
                          IsErrorReq(prim)  or IsRejectReq(prim) or
                          IsEventReportReq(prim) or IsNotificationReq(prim);
  IsInd(prim)           = IsInvokeInd(prim) or IsResultInd(prim) or
                          IsErrorInd(prim) or IsRejectInd(prim) or
                          IsEventReportInd(prim) or IsNotificationInd(prim);
  IsConf(prim)          = Isinvokeconf(prim);

ENDTYPE(*RSP_ServiceType*)


TYPE RSP_ServiceElements is RSP_Classifier, Boolean
OPNS
  IsInvoke          : RSP -> Bool
  IsResult          : RSP -> Bool
  IsError           : RSP -> Bool
  IsEventReport     : RSP -> Bool
  IsNotification    : RSP -> Bool
  IsReject          : RSP -> Bool
EQNS
FORALL
```

```
  prim : RSP
OFSORT bool
  IsInvoke(prim)      = IsInvokeReq(prim) or IsInvokeInd(prim) or
                        IsInvokeConf(prim);
  IsResult(prim)      = IsResultReq(prim) or IsResultInd(prim);
  IsError(prim)       = IsErrorReq(prim) or IsErrorInd(prim);
  IsEventReport(prim) = IsEventReportReq(prim) or IsEventReportInd(prim);
  IsNotification(prim)= IsNotificationReq(prim) or IsNotificationInd(prim);
  IsReject(prim)      = IsRejectInd(prim) or IsRejectReq(prim);

ENDTYPE(*RSP_ServiceElements*)


TYPE matching2
IS RSP_ServiceElements, Boolean
OPNS
  _matches2_    : RSP, RSP -> Bool
EQNS
FORALL
  p,q : RSP
OFSORT bool
  p matches2 q  = (Isinvoke(p) and Isinvoke(q)) or
                  (IsResult(p) and IsResult(q)) or
                  (IsError(p) and IsError(q)) or
                  (IsEventReport(p) and IsEventReport(q)) or
                  (IsNotification(p) and IsNotification(q)) or
                  (IsReject(p) and IsReject(q));

ENDTYPE (* matching2 *)


TYPE RSP_Relations
IS RSP_Classifier, Boolean, RSP_Attributes, ID_Number,
   RSP_Servicetype,matching2, kind, dataunitsfunctions
OPNS
  _IndForReq_       : RSP, RSP -> Bool
  _ConfForReq_      : RSP, RSP -> Bool
  _AnswerForInv_    : RSP, RSP -> Bool
  _AnswerToInv_     : RSP, RSP -> Bool

EQNS
FORALL prim1,prim2 : RSP
OFSORT Bool
  prim1 IndForReq prim2       = IsInd(prim1) and IsReq(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches2 prim2) and
                                (inv(prim1) eq inv(prim2)) and
                                (perf(prim1) eq perf(prim2)) and
                                (pdu(prim1) eq pdu(prim2));
  prim1 ConfForReq prim2      = IsConf(prim1) and IsReq(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches2 prim2) and
                                (inv(prim1) eq perf(prim2)) and
                                (inv(prim2) eq perf(prim1)) and
                                (pdu(prim1) eq pdu(prim2));
  prim1 AnswerForInv prim2    = (GetId(prim1) eq GetId(prim2)) and
                                (IsErrorReq(prim1) or IsRejectReq(prim1) or
                                 IsResultReq(prim1)) and IsInvokeInd(prim2) and
                                (inv(prim1)  eq perf(prim2)) and
                                (inv(prim2) eq perf(prim1));
  prim1 AnswerToInv prim2     = (GetId(prim1) eq GetId(prim2)) and
                                (IsErrorInd(prim1) or IsRejectInd(prim1) or
                                 IsResultInd(prim1)) and IsInvokeReq(prim2) and
                                (inv(prim1) eq perf(prim2)) and
                                (inv(prim2) eq perf(prim1));
ENDTYPE(*RSP_Relations*)

TYPE kind IS
SORTS
  kind
OPNS
  req   : -> kind
  ind   : -> kind
  res   : -> kind
  conf  : -> kind
ENDTYPE (* kind *)


TYPE dataunits IS ID_Number, Boolean, AId
SORTS
```

```
  ompdu
OPNS
  enableresult  : aid,id,bool -> ompdu
  enable        : aid,id -> ompdu
  disable       : aid,id -> ompdu
  uset          : aid,id -> ompdu
  create        : aid,id,AI -> ompdu
  notification  : aid,id -> ompdu
  report        : aid,id -> ompdu

ENDTYPE (* dataunits *)


TYPE dataunitsfunctions IS dataunits,om_service_primitives, kind,
     boolean, OSP_Match, OSP_Classifier, ID_Number, AID
OPNS
  decode        : ompdu, kind -> osp
  code          : osp -> ompdu
  _eq_          : ompdu,ompdu -> bool
  getpduaid     : ompdu -> aid
EQNS
FORALL j,j1 :id, aa,aa1 :aid, b,b1 : bool, a,a1 : AI
OFSORT bool
  enableresult(aa,j,b) eq enableresult(aa1,j1,b1) = (aa eq aa1) and (j eq j1) and (b eq b1);
  enable(aa,j) eq enable(aa1,j1) = (aa eq aa1) and (j eq j1);
  disable(aa,j) eq disable(aa1,j1) = (aa eq aa1) and (j eq j1);
  uset(aa,j) eq uset(aa1,j1)      = (aa eq aa1) and (j eq j1);
  create(aa,j,a) eq create(aa1,j1,a1) = (aa eq aa1) and (j eq j1) and (a eq a1);
  notification(aa,j) eq notification(aa1,j1) = (aa eq aa1) and (j eq j1);
  report(aa,j) eq report(aa1,j1) = (aa eq aa1) and (j eq j1);
OFSORT aid
  getpduaid(enable(aa,j))       = aa;
  getpduaid(enableresult(aa,j,b))= aa;
  getpduaid(disable(aa,j))      = aa;
  getpduaid(uset(aa,j))         = aa;
  getpduaid(create(aa,j,a))     = aa;
  getpduaid(notification(aa,j)) = aa;
  getpduaid(report(aa,j))       = aa;
OFSORT osp
  decode(disable(aa,j),ind)     = disableind(aa,j);
  decode(disable(aa,j),req)     = disablereq(aa,j);
  decode(disable(aa,j),conf)    = disableconf(aa,j);
  decode(enable(aa,j),ind)      = enableind(aa,j);
  decode(enable(aa,j),req)      = enablereq(aa,j);
  decode(enableresult(aa,j,b),conf) = enableconf(aa,j,b);
  decode(uset(aa,j),req)        = usetreq(aa,j);
  decode(uset(aa,j),ind)        = usetind(aa,j);
  decode(create(aa,j,a),req)    = createreq(aa,j,a);
  decode(create(aa,j,a),ind)    = createind(aa,j,a);
  decode(create(aa,j,a),res)    = createres(aa,j,a);
  decode(create(aa,j,a),conf)   = createconf(aa,j,a);
  decode(notification(aa,j),req)= notificationreq(aa,j);
  decode(notification(aa,j),ind)= notificationind(aa,j);
  decode(report(aa,j),req)      = reportreq(aa,j);
  decode(report(aa,j),ind)      = reportind(aa,j);
OFSORT ompdu
  code(disableind(aa,j))        = disable(aa,j);
  code(disablereq(aa,j))        = disable(aa,j);
  code(disableconf(aa,j))       = disable(aa,j);
  code(enableind(aa,j))         = enable(aa,j);
  code(enablereq(aa,j))         = enable(aa,j);
  code(enableconf(aa,j,b))      = enableresult(aa,j,b);
  code(usetreq(aa,j))           = uset(aa,j);
  code(usetind(aa,j))           = uset(aa,j);
  code(createreq(aa,j,a))       = create(aa,j,a);
  code(createind(aa,j,a))       = create(aa,j,a);
  code(createres(aa,j,a))       = create(aa,j,a);
  code(createconf(aa,j,a))      = create(aa,j,a);
  code(notificationreq(aa,j))   = notification(aa,j);
  code(notificationind(aa,j))   = notification(aa,j);
  code(reportreq(aa,j))         = report(aa,j);
  code(reportind(aa,j))         = report(aa,j);
ENDTYPE (* dataunits *)


TYPE fifoqueue IS boolean
FORMALSORTS data
SORTS queue
OPNS
  null          : -> queue
  add           : data,queue -> queue
  head          : queue -> data
  tail          : queue -> queue
  isempty       : queue -> bool

EQNS
FORALL  d1,d2 : data, q,r : queue
OFSORT queue
  tail(null)                    = null;
  tail(add(d1,null))            = null;
  tail(add(d1,add(d2,q)))       = add(d1,tail(add(d2,q)));
OFSORT data
  head(add(d1,null))            = d1;
  head(add(d1,add(d2,q)))       = head(add(d2,q));
OFSORT bool
  isempty(null)                 = true;
  isempty(add(d1,q))            = false;
ENDTYPE (* fifoqueue *)


TYPE pduqueue IS fifoqueue ACTUALIZEDBY dataunits USING
SORTNAMES
  ompdu FOR data
  pduqueue FOR queue
ENDTYPE (* pduqueue *)


TYPE Address_Identifier
IS Boolean
SORTS AI
OPNS offset            : -> AI
     neighbour         : AI -> AI
     _eq_,_lt_         : AI, AI -> Bool
EQNS
FORALL
  x,y : AI
OFSORT Bool
       offset eq offset                    = true;
       offset eq neighbour(x)              = false;
       neighbour(x) eq offset              = false;
       neighbour(x) eq neighbour(y)        = x eq y;
       offset lt offset                    = false;
       offset lt neighbour(x)              = true;
       neighbour(x) lt offset              = false;
       neighbour(x) lt neighbour(y)        = x lt y;
ENDTYPE(*Address_Identifier*)


TYPE ID_Number
IS Boolean
SORTS ID
OPNS 0                 : -> ID
     succ              : ID -> ID
     _eq_,_lt_,_ne_    : ID, ID -> Bool
EQNS
FORALL
  x,y : ID
OFSORT Bool
       0 eq 0                              = true;
       0 eq succ(x)                        = false;
       succ(x) eq 0                        = false;
       succ(x) eq succ(y)                  = x eq y;
       x ne y                              = not(x eq y);
       0 lt 0                              = false;
       0 lt succ(x)                        = true;
       succ(x) lt 0                        = false;
       succ(x) lt succ(y)                  = x lt y;
ENDTYPE(*ID_Number*)


TYPE IDSet
IS set ACTUALIZEDBY ID_Number USING
SORTNAMES ID FOR Element
          IDSet FOR Set
          Bool FOR FBool
OPNNAMES noid FOR {}
ENDTYPE (* IDSet *)


TYPE AId
```

```
IS Address_Identifier, ID_Number, Boolean
SORTS AId
OPNS
  AId           : AI, AI -> AId
  Initiator     : AId -> AI
  Responder     : AId -> AI
  _eq_,_ne_,_lt_: AId, AId -> Bool
EQNS
FORALL
  a1,a2 : AI, as1,as2 : AId
OFSORT AI
       Initiator(AId(a1,a2))         = a1;
       Responder(AId(a1,a2))         = a2;
OFSORT Bool
       as1 eq as2  = (Initiator(as1) eq Initiator(as2)) and
                     (Responder(as1) eq Responder(as2));
       as1 ne as2  = not(as1 eq as2);
       as1 lt as2  = (Initiator(as1) lt Initiator(as2)) or
                     ((Initiator(as1) eq Initiator(as2)) and
                     (Responder(as1) lt Responder(as2)));
ENDTYPE(*AId*)

TYPE AssociationSet
IS set ACTUALIZEDBY AId USING
SORTNAMES
  AId FOR Element
  AIdSet FOR Set
  Bool  FOR FBool
OPNNAMES
  empty FOR {}
ENDTYPE(*AssociationSet*)


TYPE OM_Service_Primitives
IS Address_Identifier, ID_Number, AId
SORTS OSP
OPNS usetreq           : AID, ID -> OSP
     enablereq         : AID, ID -> OSP
     disablereq        : AID, ID -> OSP
     reportreq         : AID, ID -> OSP
     notificationreq   : AID, ID -> OSP
     createreq         : AID, ID, AI -> OSP
     usetind           : AID, ID -> OSP
     enableind         : AID, ID -> OSP
     disableind        : AID, ID -> OSP
     reportind         : AID, ID -> OSP
     notificationind   : AID, ID -> OSP
     createind         : AID, ID, AI -> OSP
     createres         : AID, ID, AI -> OSP
     enableconf        : AID, ID, Bool -> OSP
     disableconf       : AID, ID -> OSP
     createconf        : AID, ID, AI -> OSP
ENDTYPE(* OM_Service_Primitives*)


TYPE Doublet
IS Boolean, NaturalNumber
SORTS Tuplet
OPNS One, Two              : -> Tuplet
     _eq_, _ne_, _lt_      : Tuplet, Tuplet -> Bool
     h                     : Tuplet -> Nat
EQNS
FORALL x,y : Tuplet
OFSORT Nat
       h(One) = 0;
       h(Two) = succ(h(one));
OFSORT Bool
       x eq y = h(x) eq h(y);
       x ne y = h(x) ne h(y);
       x lt y = h(x) lt h(y);
ENDTYPE (*Doublet*)


TYPE SixteenTuplet
IS Doublet
OPNS Three, Four, Five, Six, Seven, Eight, Nine, Ten,
     Eleven, Twelve, Thirteen, Fourteen, Fifteen, Sixteen :-> Tuplet
EQNS
OFSORT Nat
       h(Three)      = succ(h(Two));
       h(Four)       = succ(h(Three));
       h(Five)       = succ(h(Four));
       h(Six)        = succ(h(Five));
       h(Seven)      = succ(h(Six));
       h(Eight)      = succ(h(Seven));
       h(Nine)       = succ(h(Eight));
       h(Ten)        = succ(h(Nine));
       h(Eleven)     = succ(h(Ten));
       h(Twelve)     = succ(h(Eleven));
       h(Thirteen)   = succ(h(Twelve));
       h(Fourteen)   = succ(h(Thirteen));
       h(Fifteen)    = succ(h(Fourteen));
       h(Sixteen)    = succ(h(Fifteen));
ENDTYPE (*SixteenTuplet*)


TYPE OSPCLASS IS SixteenTuplet RENAMEDBY
SORTNAMES
  OSPCLASS FOR Tuplet
OPNNAMES
    USetRequest                 FOR One
    EnableRequest               FOR Two
    DisableRequest              FOR Three
    ReportRequest               FOR Four
    NotificationRequest         FOR Five
    CreateRequest               FOR Six
    USetIndication              FOR Seven
    EnableIndication            FOR Eight
    DisableIndication           FOR Nine
    ReportIndication            FOR Ten
    NotificationIndication      FOR Eleven
    CreateIndication            FOR Twelve
    CreateResponse              FOR Thirteen
    EnableConfirmation          FOR Fourteen
    DisableConfirmation         FOR Fifteen
    CreateConfirmation          FOR Sixteen
ENDTYPE(* OSPCLASS *)


TYPE OSP_Classifier
IS OSPClass, OM_Service_Primitives, AId
OPNS Map              : OSP -> OSPClass
     IsUsetReq        : OSP -> Bool
     IsEnableReq      : OSP -> Bool
     IsDisableReq     : OSP -> Bool
     IsReportReq      : OSP -> Bool
     IsNotificationReq : OSP -> Bool
     IsCreateReq      : OSP -> Bool
     IsUsetInd        : OSP -> Bool
     IsEnableInd      : OSP -> Bool
     IsDisableInd     : OSP -> Bool
     IsReportInd      : OSP -> Bool
     IsNotificationInd : OSP -> Bool
     IsCreateInd      : OSP -> Bool
     IsCreateRes      : OSP -> Bool
     IsEnableConf     : OSP -> Bool
     IsDisableConf    : OSP -> Bool
     IsCreateConf     : OSP -> Bool
EQNS
FORALL
  a : AI, as1,as : AId, j : ID, prim : OSP, b : Bool
OFSORT OSPClass
       Map(USetReq(as,j))              = USetRequest;
       Map(EnableReq(as,j))            = EnableRequest;
       Map(DisableReq(as,j))           = DisableRequest;
       Map(ReportReq(as,j))            = ReportRequest;
       Map(NotificationReq(as,j))      = NotificationRequest;
       Map(CreateReq(as,j,a))          = CreateRequest;
       Map(USetInd(as,j))              = USetIndication;
       Map(EnableInd(as,j))            = EnableIndication;
       Map(DisableInd(as,j))           = DisableIndication;
       Map(ReportInd(as,j))            = ReportIndication;
       Map(NotificationInd(as,j))      = NotificationIndication;
       Map(CreateInd(as,j,a))          = CreateIndication;
       Map(CreateRes(as,j,a))          = CreateResponse;
       Map(EnableConf(as,j,b))         = EnableConfirmation;
       Map(DisableConf(as,j))          = DisableConfirmation;
       Map(CreateConf(as,j,a))         = CreateConfirmation;
OFSORT Bool
       IsUsetReq(prim)                   = map(prim) eq USetRequest;
       IsEnableReq(prim)                 = map(prim) eq EnableRequest;
       IsDisableReq(prim)                = map(prim) eq DisableRequest;
       IsReportReq(prim)                 = map(prim) eq ReportRequest;
       IsNotificationReq(prim)           = map(prim) eq NotificationRequest;
       IsCreateReq(prim)                 = map(prim) eq CreateRequest;
       IsUsetInd(prim)                   = map(prim) eq USetIndication;
       IsEnableInd(prim)                 = map(prim) eq EnableIndication;
       IsDisableInd(prim)                = map(prim) eq DisableIndication;
       IsReportInd(prim)                 = map(prim) eq ReportIndication;
       IsNotificationInd(prim)           = map(prim) eq NotificationIndication;
       IsCreateInd(prim)                 = map(prim) eq CreateIndication;
       IsCreateRes(prim)                 = map(prim) eq CreateResponse;
       IsEnableConf(prim)                = map(prim) eq EnableConfirmation;
       IsDisableConf(prim)               = map(prim) eq DisableConfirmation;
       IsCreateConf(prim)                = map(prim) eq CreateConfirmation;
ENDTYPE(*OSP_Classifier*)


TYPE OSP_Attributes
IS OSP_Classifier, AId, Boolean
OPNS GetId                : OSP -> ID
     Subj                 : OSP -> AI
     GetAId               : OSP -> AId
     GetSuccess           : OSP -> Bool
EQNS
FORALL
  a : AI, j : ID, as : AId, b : Bool
OFSORT Bool
  GetSuccess(EnableConf(as,j,b)) = b;
OFSORT ID
  GetId(USetReq(as,j))            = j;
  GetId(EnableReq(as,j))          = j;
  GetId(DisableReq(as,j))         = j;
  GetId(ReportReq(as,j))          = j;
  GetId(NotificationReq(as,j))    = j;
  GetId(CreateReq(as,j,a))        = j;
  GetId(USetInd(as,j))            = j;
  GetId(EnableInd(as,j))          = j;
  GetId(DisableInd(as,j))         = j;
  GetId(ReportInd(as,j))          = j;
  GetId(NotificationInd(as,j))    = j;
  GetId(CreateInd(as,j,a))        = j;
  GetId(CreateRes(as,j,a))        = j;
  GetId(EnableConf(as,j,b))       = j;
  GetId(DisableConf(as,j))        = j;
  GetId(CreateConf(as,j,a))       = j;
OFSORT AI
  Subj(CreateReq(as,j,a))        = a;
  Subj(CreateInd(as,j,a))        = a;
  Subj(CreateRes(as,j,a))        = a;
  Subj(CreateConf(as,j,a))       = a;
OFSORT AId
  GetAID(USetReq(as,j))            = as;
  GetAID(EnableReq(as,j))          = as;
  GetAID(DisableReq(as,j))         = as;
  GetAID(ReportReq(as,j))          = as;
  GetAID(NotificationReq(as,j))    = as;
  GetAID(CreateReq(as,j,a))        = as;
  GetAID(USetInd(as,j))            = as;
  GetAID(EnableInd(as,j))          = as;
  GetAID(DisableInd(as,j))         = as;
  GetAID(ReportInd(as,j))          = as;
  GetAID(NotificationInd(as,j))    = as;
  GetAID(CreateInd(as,j,a))        = as;
  GetAID(CreateRes(as,j,a))        = as;
  GetAID(EnableConf(as,j,b))       = as;
  GetAID(DisableConf(as,j))        = as;
  GetAID(CreateConf(as,j,a))       = as;
ENDTYPE(*OSP_Attributes*)


TYPE OSP_Servicetype
IS OSP_Classifier

OPNS
  IsReq                : OSP -> Bool
  IsInd                : OSP -> Bool
  IsRes                : OSP -> Bool
  IsConf               : OSP -> Bool
```

```
EQNS
FORALL
  prim : OSP
OFSORT bool

  IsReq(prim)           = IsUSetReq(prim) or IsEnableReq(prim) or
                          IsDisableReq(prim) or IsCreateReq(prim) or
                          IsReportReq(prim) or IsNotificationReq(prim);
  IsInd(prim)           = IsUSetInd(prim) or IsEnableInd(prim) or
                          IsDisableInd(prim) or IsCreateInd(prim)  or
                          IsReportInd(prim) or IsNotificationInd(prim);
  IsRes(prim)           = IsCreateRes(prim);
  IsConf(prim)          = IsEnableConf(prim) or IsDisableConf(prim) or
                          IsCreateConf(prim);

ENDTYPE(*OSP_ServiceType*)


TYPE OSP_Elements is OSP_Classifier, Boolean
OPNS
  IsUSet            : OSP -> Bool
  IsEnable          : OSP -> Bool
  IsDisable         : OSP -> Bool
  IsReport          : OSP -> Bool
  IsNotification    : OSP -> Bool
  IsCreate          : OSP -> Bool
EQNS
FORALL
  prim : OSP
OFSORT bool
  IsUSet(prim)        = IsUSetReq(prim) or IsUSetInd(prim);
  IsEnable(prim)      = IsEnableReq(prim) or IsEnableInd(prim) or
                        IsEnableConf(prim);
  IsDisable(prim)     = IsDisableReq(prim) or IsDisableInd(prim) or
                        IsDisableConf(prim);
  IsReport(prim)      = IsReportReq(prim) or IsReportInd(prim);
  IsNotification(prim)= IsNotificationReq(prim) or IsNotificationInd(prim);
  IsCreate(prim)      = IsCreateReq(prim) or IsCreateInd(prim) or
                        IsCreateRes(prim) or IsCreateConf(prim);
ENDTYPE(*OSP_Elements*)


TYPE OSP_Type
IS
  Boolean, OSP_Elements

OPNS

  IsUnConf          : OSP -> Bool
  IsUserConf        : OSP -> Bool
  IsProviderConf    : OSP -> Bool

EQNS
FORALL prim : OSP

OFSORT Bool
  IsUnConf(prim)        = IsUSet(prim) or IsReport(prim) or
                          IsNotification(prim);
  IsUserConf(prim)      = IsCreate(prim);
  IsProviderConf(prim)  = IsEnable(prim) or IsDisable(prim);
ENDTYPE (* OSP_Type *)


TYPE OSP_Match
IS OSP_Elements, Boolean
OPNS
  _matches_         : OSP, OSP -> Bool
EQNS
FORALL p,q : OSP
OFSORT Bool
  p matches q       = (IsDisable(p) and IsDisable(q)) or
                      (IsEnable(p) and IsEnable(q)) or
                      (IsUset(p) and IsUset(q)) or
                      (IsCreate(p) and IsCreate(q)) or
                      (IsReport(p) and IsReport(q)) or
                      (IsNotification(p) and IsNotification(q));

ENDTYPE (* OSP_Match *)
```

```
TYPE OSP_Primitives_Relations
IS OSP_Classifier, Boolean, OSP_Attributes, ID_Number,OSP_Servicetype,
   OSP_Match
OPNS
  _ConfForReq_      : OSP, OSP -> Bool
  _ConfForRes_      : OSP, OSP -> Bool
  _IndForReq_       : OSP, OSP -> Bool
  _ResForInd_       : OSP, OSP -> Bool
EQNS
FORALL prim1,prim2 : OSP
OFSORT Bool
  prim1 ConfForReq prim2      = IsConf(prim1) and IsReq(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);
  prim1 ConfForRes prim2      = IsConf(prim1) and IsRes(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);
  prim1 IndForReq prim2       = IsInd(prim1) and IsReq(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);
  prim1 ResForInd prim2       = IsRes(prim1) and IsInd(prim2) and
                                (GetId(prim1) eq GetId(prim2)) and
                                (prim1 matches prim2);
ENDTYPE(*OSP_Primitives_Relations*)


BEHAVIOUR

  HIDE RSAP IN (
  (
  omampe[OSAP,RSAP](insert(aid(offset,neighbour(offset)),empty), neighbour(offset))
|||
  omampe[OSAP,RSAP](insert(aid(offset,neighbour(offset)),empty),offset)
  )
|[RSAP]|
  communications[RSAP] )

WHERE



  PROCESS communications[RSAP] : NOEXIT :=

    communication[RSAP]
  |||
    i;
    communications[RSAP]

  ENDPROC (* communications *)


  PROCESS communication[RSAP] : EXIT :=

    (invoker[RSAP] ||| performer[RSAP])
  ||
    syncip[RSAP]

  ENDPROC (* communication *)


    PROCESS invoker[RSAP] : EXIT :=

      RSAP ?n : AI ?p1: RSP [IsInvokeReq(p1)];
      ( RSAP !n ?p2: RSP [(p2 ConfForReq p1) and IsAck(p2)];
        RSAP !n ?p2: RSP [(p2 AnswerToInv p1)];
        EXIT
      []
        i; (* TimeOut *)
        RSAP !n ?p2: RSP [(p2 ConfForReq p1) and IsNack(p2)];
        EXIT
      )
    []
      RSAP ?n : AI ?p1: RSP [IsNotificationReq(p1) or IsEventReportReq(p1)];
      EXIT

    ENDPROC (* invoker *)


    PROCESS performer[RSAP] : EXIT :=
      RSAP ?n : AI ?p1: RSP [IsInvokeInd(p1)];
      RSAP !n  ?p2: RSP [p2 AnswerForInv p1];
      EXIT
    []
      RSAP ?n : AI ?p1: RSP[IsNotificationInd(p1) or IsEventReportInd(p1)];
      EXIT

    ENDPROC (* performer *)


    PROCESS syncip[RSAP] : EXIT :=

      RSAP ?n : AI ?p1: RSP [IsInvokeReq(p1)];
      ( RSAP !n ?p2: RSP [IsInvokeConf(p2) and IsNack(p2)];
        EXIT
      []
        RSAP !perf(p1) ?p2: RSP [p2 IndForReq p1];
        RSAP !n ?p2: RSP [IsInvokeConf(p2) and IsAck(p2)];
        RSAP !perf(p1) ?p3: RSP [IsReq(p3) and
                                 (IsError(p3) or IsReject(p3) or IsResult(p3))];
        RSAP !n ?p4: RSP [p4 IndForReq p3];
        EXIT )
    []
      RSAP ?n : AI ?p1: RSP[IsEventReportReq(p1) or IsNotificationReq(p1)];
      RSAP !perf(p1) ?p2: RSP[p2 IndForReq p1];
      EXIT

    ENDPROC (* syncip *)



PROCESS omampe[OSAP,RSAP](A : aidset, n : AI) : NOEXIT :=

  associations[OSAP,RSAP](A,n)

ENDPROC (* omampe *)

PROCESS associations[OSAP,RSAP](A : aidset, n : AI) : NOEXIT :=

  CHOICE aa : aid [] [aa IsIn A] ->
  (i; associations[OSAP,RSAP](remove(aa,A),n) ||| association[OSAP,RSAP](aa,n) )

ENDPROC (* associations *)


PROCESS association[OSAP,RSAP](aa : aid, n : AI) : NOEXIT :=

  HIDE INIT,CON,UNC,MNS,REL,ABORT IN
    (upfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n) |[OSAP]| omsi[OSAP](aa,n) )

  |[INIT,CON,UNC,MNS,REL,ABORT]|

    (lpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n) |[RSAP]| rrsi[RSAP](aa,n) )

ENDPROC (* association *)


PROCESS omsi[OSAP](aa : aid, n : ai) : NOEXIT :=

[n eq initiator(aa)] -> manager[OSAP](aa)
[]
[n eq responder(aa)] -> agent[OSAP](aa)

ENDPROC (* omsi *)


PROCESS rrsi[RSAP](aa : aid, n : ai) : NOEXIT :=

(invoker[RSAP] ||| performer[RSAP])
|||
i;rrsi[RSAP](aa,n)

ENDPROC (* rrsi *)


PROCESS upfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI): noexit :=

  [n eq initiator(aa)] -> mupfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  [n eq responder(aa)] -> aupfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)
```

```
ENDPROC (* upfs *)


PROCESS lpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI): noexit :=

  [n eq initiator(aa)] -> mlpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  [n eq responder(aa)] -> alpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)

ENDPROC (* lpfs *)


PROCESS mupfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI): NOEXIT :=

  mu_initiate[OSAP,INIT](aa,n) >>
  ( mu_communicate[OSAP,CON,UNC,MNS](aa,n) ||| mu_release[OSAP,REL](aa,n)  )
  [>
    mu_abort[OSAP,ABORT](aa,n) >> mupfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)

ENDPROC (* mupfs *)


PROCESS aupfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI): NOEXIT :=

  au_initiate[OSAP,INIT](aa,n) >>
  ( au_communicate[OSAP,CON,UNC,MNS](aa,n) ||| au_release[OSAP,REL](aa,n)  )
  [>
    au_abort[OSAP,ABORT](aa,n) >> aupfs[OSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)

ENDPROC (* aupfs *)


PROCESS mlpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI): NOEXIT :=

 (
  ml_initiate[RSAP,INIT](aa,n) >>
  (( ml_communicate[RSAP,CON,UNC,MNS,ABORT](aa,n) |[CON,UNC,MNS,ABORT]|
     ml_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n) )
  [>
    ml_abort[RSAP,ABORT](aa,n)) >> mlpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)
  )
 |[RSAP]|
   identification[RSAP](aa,n)

ENDPROC (* mlpfs *)


PROCESS alpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI): NOEXIT :=

 (
  al_initiate[RSAP,INIT](aa,n) >>
  (( al_communicate[RSAP,CON,UNC,MNS,ABORT](aa,n) |[CON,UNC,MNS,ABORT]|
    al_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n)  )
  [>
    al_abort[RSAP,ABORT](aa,n)) >> alpfs[RSAP,INIT,CON,UNC,MNS,REL,ABORT](aa,n)
  )
 |[RSAP]|
   identification[RSAP](aa,n)

ENDPROC (* alpfs *)
PROCESS  mu_initiate[OSAP,INIT](aa : aid, n : AI) : EXIT :=

  OSAP !n ?p : OSP [IsEnableReq(p) and (aa eq getaid(p))];
  INIT !aa !getid(p) !code(p);
  INIT !aa !getid(p) ?pp : ompdu;
  OSAP !n !decode(pp,conf);
( [not(getsuccess(decode(pp,conf)))] -> mu_initiate[OSAP,INIT](aa,n)
[]
  [getsuccess(decode(pp,conf))     ] -> EXIT )

ENDPROC (* mu_initiate *)


PROCESS  au_initiate[OSAP,INIT](aa : aid, n : AI) : EXIT :=

  INIT !aa ?j : id ?p : ompdu;
  OSAP !n !decode(p,ind);
  INIT !aa !j !code(enableconf(aa,j,true));
  EXIT
```

```
ENDPROC (* au_initiate *)


PROCESS  mu_communicate[OSAP,CON,UNC,MNS](aa : aid, n : AI) : NOEXIT :=

  mu_con[OSAP,CON](aa, n, null)
|||
  mu_unc[OSAP,UNC](aa, n)
|||
  mu_mns[OSAP,MNS](aa, n)

WHERE

  PROCESS  mu_con[OSAP,CON](aa : aid, n : AI, q : pduqueue) : NOEXIT :=

    OSAP !n ?p : OSP[IsUserConf(p) and IsReq(p) and (aa eq getaid(p))];
    mu_con[OSAP,CON](aa,n, add(code(p),q))
  []
    [not(isempty(q))] -> CON !aa !getid(decode(head(q),req)) !head(q) !req;
                         mu_con[OSAP,CON](aa,n,tail(q))
  []
    CON !aa ?j : ID ?p : ompdu !conf;
    OSAP !n !decode(p,conf);
    mu_con[OSAP,CON](aa,n,q)

  ENDPROC (* mu_con *)

  PROCESS  mu_unc[OSAP,UNC](aa : aid, n : AI) : NOEXIT :=

    OSAP !n ?p : OSP[IsUSet(p) and (aa eq getaid(p)) and IsReq(p)];
    UNC !aa !getid(p) !code(p) !req;
    EXIT
  |||
    i;
    mu_unc[OSAP,UNC](aa,n)

  ENDPROC (* mu_unc *)

  PROCESS  mu_mns[OSAP,MNS](aa : aid, n : AI) : NOEXIT :=

    MNS !aa ?j : ID ?p : ompdu !ind;
    OSAP !n !decode(p,ind);
    EXIT
  |||
    OSAP !n ?p : OSP[IsNotificationReq(p) and (aa eq getaid(p))];
    MNS !aa !getid(p) !code(p) !req;
    EXIT
  |||
    i;
    mu_mns[OSAP,MNS](aa,n)

  ENDPROC (* mu_mns *)

ENDPROC (* mu_communicate *)


PROCESS  au_communicate[OSAP,CON,UNC,MNS](aa : aid, n : AI) : NOEXIT :=

  au_con[OSAP,CON](aa, n)
|||
  au_unc[OSAP,UNC](aa, n)
|||
  au_mns[OSAP,MNS](aa, n)

WHERE

  PROCESS  au_con[OSAP,CON](aa : aid, n : AI) : NOEXIT :=

    CON !aa ?j : ID ?p : ompdu !ind;
    OSAP !n !decode(p,ind);
    OSAP !n ?pp : OSP[pp ResForInd decode(p,ind)];
    CON !aa !getid(pp) !code(pp) !res;
    EXIT
  |||
    i;
    au_con[OSAP,CON](aa,n)

  ENDPROC (* au_con *)
  PROCESS  au_unc[OSAP,UNC](aa : aid, n : AI) : NOEXIT :=

    UNC !aa ?j : ID ?p : ompdu !ind;
    OSAP !n !decode(p,ind);
    EXIT
  |||
    i;
    au_unc[OSAP,UNC](aa,n)

  ENDPROC (* au_unc *)

  PROCESS  au_mns[OSAP,MNS](aa : aid, n : AI) : NOEXIT :=

    MNS !aa ?j : ID ?p : ompdu !ind;
    OSAP !n !decode(p,ind);
    EXIT
  |||
    OSAP !n ?p : OSP[(IsNotificationReq(p) or IsReportReq(p)) and (aa eq getaid(p))];
    MNS !aa !getid(p) !code(p) !req;
    EXIT
  |||
    i;
    au_mns[OSAP,MNS](aa,n)

  ENDPROC (* au_unc *)

ENDPROC (* au_communicate *)


PROCESS  mu_release[OSAP,REL](aa : aid, n : AI) : NOEXIT :=

  OSAP !n ?p : OSP [IsDisableReq(p) and (aa eq getaid(p))];
  REL !aa !getid(p) !code(p);
  STOP

ENDPROC (* mu_release *)

PROCESS  au_release[OSAP,REL](aa : aid, n : AI) : NOEXIT :=

  OSAP !n ?p : OSP [IsDisableReq(p) and (aa eq getaid(p))];
  REL !aa !getid(p) !code(p);
  STOP

ENDPROC (* au_release *)


PROCESS mu_abort[OSAP,ABORT](aa : aid, n : AI) : EXIT :=

  ABORT !aa ?j : ID ?p : ompdu ?k : kind;
  OSAP !n !decode(p,k);
  EXIT

ENDPROC (* mu_abort *)


PROCESS au_abort[OSAP,ABORT](aa : aid, n : AI) : EXIT :=

  ABORT !aa ?j : ID ?p : ompdu ?k : kind;
  OSAP !n !decode(p,k);
  EXIT

ENDPROC (* au_abort *)


PROCESS identification[RSAP](aa : aid,n : ai) : NOEXIT :=

  RSAP ?n : AI ?p : RSP[getpduaid(pdu(p)) eq aa];
  identification[RSAP](aa,n)

ENDPROC (* identification *)


PROCESS ml_initiate[RSAP,INIT](aa : aid, n : AI) : EXIT :=
  ml_idle[RSAP](aa,n)
[>
  INIT !aa ?j : ID ?p: ompdu;
  RSAP !n !invokereq(n,responder(aa),j,p);
  filter[RSAP](aa,n,j) >> ACCEPT p : ompdu, j : id IN
  [getsuccess(decode(p,conf))] -> INIT !aa !j !p; EXIT
[]
```

```
  [not(getsuccess(decode(p,conf)))] -> INIT !aa !j !p; ml_initiate[RSAP,INIT](aa,n)

WHERE

  PROCESS ml_idle[RSAP](aa : aid, n : AI) : NOEXIT :=

    RSAP !n ?p : RSP [IsInvokeInd(p)];
    RSAP !n !rejectreq(n,responder(aa),getid(p),pdu(p));
    EXIT
  |||
    RSAP !n ?p : RSP[not(IsInvokeInd(p) or IsInvokeReq(p))];
    EXIT
  |||
    i;
    ml_idle[RSAP](aa,n)

  ENDPROC (* ml_idle *)

PROCESS filter[RSAP](aa : aid, n : AI, j : id) : EXIT(ompdu,id) :=

  ml_waiting1[RSAP](aa,n,j)
[>
  RSAP !n ?p : RSP[(j eq GetId(p)) and IsInvokeConf(p)];
( [isnack(p)] ->
    EXIT(code(enableconf(aa,j,false)),j)
[]
  [isack(p) ] ->
    (ml_waiting2[RSAP](aa,n,j)
  [>
    RSAP !n ?p : RSP[(j eq GetId(p)) and (IsErrorInd(p) or
                     IsResultInd(p) or IsRejectInd(p))];
    EXIT(pdu(p),j))
  )

WHERE

    PROCESS ml_waiting1[RSAP](aa : aid, n : AI, j : id) : NOEXIT :=

      RSAP !n ?p : RSP [not(IsInvokeInd(p) or
                          (IsInvokeConf(p) and (j eq getid(p))))];
      EXIT
    |||
      RSAP !n ?p : RSP [IsInvokeInd(p)];
      RSAP !n !rejectreq(n,responder(aa),getid(p),pdu(p));
      EXIT
    |||
      i;
      ml_waiting1[RSAP](aa,n,j)

    ENDPROC (* ml_waiting1 *)

    PROCESS ml_waiting2[RSAP](aa : aid, n : AI, j : id) : NOEXIT :=

      RSAP !n ?p : RSP [not(IsInvokeInd(p) or
                           ((IsErrorInd(p) or IsResultInd(p) or
                             IsRejectInd(p)) and (j eq getid(p))))];
      EXIT
    |||
      RSAP !n ?p : RSP [IsInvokeInd(p)];
      RSAP !n !rejectreq(n,responder(aa),getid(p),pdu(p));
      EXIT
    |||
      i;
      ml_waiting2[RSAP](aa,n,j)

    ENDPROC (* ml_waiting2 *)

  ENDPROC (* filter *)

ENDPROC (* ml_initiate *)


PROCESS  al_initiate[RSAP,INIT](aa : aid, n : AI) : EXIT :=

  al_idle[RSAP](aa,n)
[>
( RSAP !n ?p : RSP[Isinvokeind(p) and isenable(decode(pdu(p),ind))];
  INIT !aa !getid(p) !pdu(p);
  INIT !aa !getid(p) ?pp : ompdu;
  RSAP !n !resultreq(n,initiator(aa),getid(p),pp);
  EXIT
)

WHERE

  PROCESS al_idle[RSAP](aa : aid, n : AI) : NOEXIT :=

    RSAP !n ?p : RSP[not(IsInvokeInd(p) and IsEnable(decode(pdu(p),ind)))];
    EXIT
  |||
    RSAP !n ?p : RSP[IsInvokeInd(p) and not(IsEnable(decode(pdu(p),ind)))];
    RSAP !n !rejectreq(n,initiator(aa),getid(p),pdu(p));
    EXIT
  |||
    i;
    al_idle[RSAP](aa,n)

  ENDPROC (* al_idle *)

ENDPROC (* al_initiate *)


PROCESS ml_communicate[RSAP,CON,UNC,MNS,ABORT](aa : aid, n : AI) : EXIT :=

(
  ml_con[RSAP,CON](aa,n,noid)
|||
  ml_unc[RSAP,UNC](aa,n)
|||
  ml_mns[RSAP,MNS](aa,n)
|||
  ml_confirmations[RSAP](aa,n)
)
[> ABORT !aa ?j : ID ?p : ompdu !conf; EXIT

WHERE

  PROCESS ml_confirmations[RSAP](aa : aid, n : AI) : NOEXIT :=

    RSAP !n ?p : RSP[IsInvokeConf(p) and IsAck(p)];
    ml_confirmations[RSAP](aa,n)

  ENDPROC (* ml_confirmations *)

  PROCESS ml_con[RSAP,CON](aa : aid, n : AI, ids : idset) : NOEXIT :=

    CON !aa ?j : ID ?p : ompdu !req;
    RSAP !n !invokereq(n,responder(aa),j,p);
    ml_con[RSAP,CON](aa,n,insert(j,ids))
  []
    RSAP !n ?p : RSP[(IsErrorInd(p) or IsResultInd(p)) and
                     (getid(p) IsIn ids)];
    CON !aa !getid(p) !pdu(p) !ind;
    ml_con[RSAP,CON](aa,n,remove(getid(p),ids))
  []
    RSAP !n ?p : RSP[(IsErrorInd(p) or IsResultInd(p)) and
                    not((getid(p) IsIn ids) or IsProviderConf(decode(pdu(p),ind)))];
    ml_con[RSAP,CON](aa,n,ids)

  ENDPROC

  PROCESS ml_unc[RSAP,UNC](aa: aid, n : AI) : NOEXIT :=

    UNC !aa ?j : ID ?p : ompdu !req;
    RSAP !n !invokereq(n,responder(aa),j,p);
    EXIT
  |||
    i;
    ml_unc[RSAP,UNC](aa,n)

  ENDPROC (* ml_unc *)

  PROCESS ml_mns[RSAP,MNS](aa: aid, n : AI) : NOEXIT :=

    RSAP !n ?p : RSP[IsnotificationInd(p) or IsEventReportInd(p)];
    MNS !aa !getid(p) !pdu(p) !ind;
    EXIT
  |||
    MNS !aa ?j : ID ?p : ompdu !req;
    RSAP !n !notificationreq(n,responder(aa),j,p);
    EXIT
  |||
    i;
    ml_mns[RSAP,MNS](aa,n)

  ENDPROC (* ml_mns *)

ENDPROC (* ml_communicate *)


PROCESS al_communicate[RSAP,CON,UNC,MNS,ABORT](aa : aid, n : AI) : EXIT :=

(
  al_con[RSAP,CON](aa,n)
|||
  al_unc[RSAP,UNC](aa,n)
|||
  al_mns[RSAP,MNS](aa,n)
)
[> ABORT !aa ?j : ID ?p : ompdu ?k : kind; EXIT

WHERE

  PROCESS al_con[RSAP,CON](aa : aid, n : AI) : NOEXIT :=

    RSAP !n ?p : RSP[Isinvokeind(p) and IsUserConf(decode(pdu(p),ind))];
    CON !aa !getid(p) !pdu(p) !ind;
    CON !aa !getid(p) ?pp : ompdu !res;
    ( RSAP !n !resultreq(n,initiator(aa),getid(p),pp);
      EXIT
    []
      RSAP !n !errorreq(n,initiator(aa),getid(p),pp);
      EXIT
    )
  |||
    i;
    al_con[RSAP,CON](aa,n)

  ENDPROC

  PROCESS al_unc[RSAP,UNC](aa: aid, n : AI) : NOEXIT :=

    RSAP !n ?p : RSP[Isinvokeind(p) and IsUSet(decode(pdu(p),ind))];
    UNC !aa !getid(p) !pdu(p) !ind;
    RSAP !n !resultreq(n,initiator(aa),getid(decode(pdu(p),ind)),pdu(p));
    EXIT
  |||
    i;
    al_unc[RSAP,UNC](aa,n)

  ENDPROC (* al_unc *)

  PROCESS al_mns[RSAP,MNS](aa : aid, n : AI) : NOEXIT :=

    RSAP !n ?p : RSP[IsnotificationInd(p)];
    MNS !aa !getid(p) !pdu(p) !ind;
    EXIT
  |||
    MNS !aa ?j : ID ?p : ompdu !req [IsNotification(decode(p,req))];
    RSAP !n !notificationreq(n,initiator(aa),j,p);
    EXIT
  |||
    MNS !aa ?j : ID ?p : ompdu !req [IsReport(decode(p,req))];
    RSAP !n !eventreportreq(n,initiator(aa),j,p);
    EXIT
  |||
    i;
    al_mns[RSAP,MNS](aa,n)

  ENDPROC (* al_unc *)

ENDPROC (* al_communicate *)


PROCESS ml_release[RSAP,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI) : EXIT :=

  CON !aa ?j : ID ?p : ompdu ?k : kind;
  ml_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  UNC !aa ?j : ID ?p : ompdu ?k : kind;
  ml_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  MNS !aa ?j : ID ?p : ompdu ?k : kind;
  ml_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  REL !aa ?j : ID ?p : ompdu;
  RSAP !n !invokereq(n,responder(aa),j,p);
  ml_releasing[RSAP,CON,ABORT](aa,n,j)

WHERE

  PROCESS ml_releasing[RSAP,CON,ABORT](aa : aid, n : AI, j : ID) : EXIT :=

    CON !aa ?j : ID ?p : ompdu ?k : kind;
    ml_releasing[RSAP,CON,ABORT](aa,n,j)
  []
    RSAP !n ?p : RSP[(IsResultInd(p) or IsErrorInd(p)) and
                     IsDisable(decode(pdu(p),conf)) and (j eq getid(p))];
    ABORT !aa !getid(p) !pdu(p) !conf;
    EXIT
  []
    RSAP !n ?p : RSP[IsInvokeConf(p) and IsAck(p)];
    ml_releasing[RSAP,CON,ABORT](aa,n,j)
  []
    RSAP !n ?p : RSP[IsInvokeInd(p) and not(IsProviderConf(decode(pdu(p),ind)))];
    RSAP !n !rejectreq(n,responder(aa),getid(p),pdu(p));
    ml_releasing[RSAP,CON,ABORT](aa,n,j)
  []
    RSAP !n ?p : RSP[(IsResultInd(p) or IsErrorInd(p)) and
                     not(IsDisable(decode(pdu(p),conf)))];
    ml_releasing[RSAP,CON,ABORT](aa,n,j)

  ENDPROC (* ml_releasing *)

ENDPROC (* ml_release *)


PROCESS al_release[RSAP,CON,UNC,MNS,REL,ABORT](aa : aid, n : AI) : EXIT :=

  CON !aa ?j : ID ?p : ompdu ?k : kind;
  al_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  UNC !aa  ?j : ID ?p : ompdu ?k : kind;
  al_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  MNS !aa ?j : ID ?p : ompdu ?k : kind;
  al_release[RSAP,CON,UNC,MNS,REL,ABORT](aa,n)
[]
  REL !aa ?j : ID ?p : ompdu[IsDisableReq(decode(p,req))];
  RSAP !n !invokereq(n,initiator(aa),j,p);
  al_releasing[RSAP,ABORT](aa,n,j)

WHERE

  PROCESS al_releasing[RSAP,ABORT](aa : aid, n : AI, j : ID) : EXIT :=

    RSAP !n ?p : RSP[(IsResultInd(p) or IsErrorInd(p)) and IsDisable(decode(pdu(p),conf)) and (j eq getid(p))];
    ABORT !aa !getid(p) !pdu(p) !conf;
    EXIT
  []
    RSAP !n ?p : RSP[IsInvokeInd(p) and not(IsProviderConf(decode(pdu(p),ind)))];
    RSAP !n !rejectreq(n,responder(aa),getid(p),pdu(p));
    al_releasing[RSAP,ABORT](aa,n,j)
  []
    RSAP !n ?p : RSP[IsInvokeConf(p) and IsAck(p)];
    al_releasing[RSAP,ABORT](aa,n,j)
  []
    RSAP !n ?p : RSP[(IsResultInd(p) or IsErrorInd(p)) and not (IsDisable(decode(pdu(p),conf)))];
    al_releasing[RSAP,ABORT](aa,n,j)

  ENDPROC (* al_releasing *)

ENDPROC (* al_release *)


PROCESS ml_abort[RSAP,ABORT](aa :aid, n : AI) : EXIT :=

  RSAP !n ?p : RSP[IsInvokeConf(p) and IsNack(p)];
  ABORT !aa !getid(p) !code(disableind(aa,getid(p))) !ind;
  EXIT
```

```
[]
  RSAP !n ?p : RSP[IsRejectInd(p)];
  ABORT !aa !getid(p) !code(disableind(aa,getid(p))) !ind;
  EXIT
[]
  RSAP !n ?p : RSP[(IsResultInd(p) or IsErrorInd(p)) and IsEnable(decode(pdu(p),conf))];
  ABORT !aa !getid(p) !code(disableind(aa,getid(p))) !ind;
  EXIT
[]
  RSAP !n ?p : RSP[IsInvokeInd(p) and IsDisable(decode(pdu(p),ind))];
  ABORT !aa !getid(p) !code(disableind(aa,getid(decode(pdu(p),ind)))) !ind;
  RSAP !n !resultreq(n,responder(aa),getid(p),code(disableconf(aa,getid(p))));
  EXIT

ENDPROC (* ml_abort *)


PROCESS al_abort[RSAP,ABORT](aa :aid, n : AI) : EXIT :=

  RSAP !n ?p : RSP[IsInvokeConf(p) and IsNack(p)];
  ABORT !aa !getid(p) !code(disableind(aa,getid(p))) !ind;
  EXIT
[]
  RSAP !n ?p : RSP[IsRejectInd(p)];
  ABORT !aa !getid(p) !code(disableind(aa,getid(p))) !ind;
  EXIT
[]
  RSAP !n ?p : RSP[IsInvokeInd(p) and IsEnable(decode(pdu(p),ind))];
  ABORT !aa !getid(p) !code(disableind(aa,getid(p))) !ind;
  EXIT
[]
  RSAP !n ?p : RSP[IsInvokeInd(p) and IsDisable(decode(pdu(p),ind))];
  ABORT !aa !getid(p) !code(disableind(aa,getid(p))) !ind;
  RSAP !n !resultreq(n,initiator(aa),getid(p),code(disableconf(aa,getid(p))));
  EXIT

ENDPROC (* ml_abort *)


  PROCESS manager[OSAP](aa : aid) : NOEXIT :=

    minitiate[OSAP](aa)
    >>
      mcommunicate[OSAP](aa)
    ||
      initiaterelease[OSAP](initiator(aa))
    [>
      (forcerelease[OSAP](initiator(aa)) >> manager[OSAP](aa) )

  ENDPROC (*manager*)


  PROCESS agent[OSAP](aa : aid) : NOEXIT :=

    ainitiate[OSAP](aa)
    >>
      acommunicate[OSAP](aa)
    ||
      initiaterelease[OSAP](responder(aa))
    [>
      (forcerelease[OSAP](responder(aa)) >> agent[OSAP](aa) )

  ENDPROC (*agent*)


  PROCESS minitiate[OSAP](aa : aid) : EXIT :=

    OSAP !initiator(aa) ?p1 : OSP [IsEnableReq(p1) and (aa eq GetAID(p1))];
    OSAP !initiator(aa) ?p2 : OSP [p2 ConfForReq p1];
    ([not(GetSuccess(p2))] -> minitiate[OSAP](aa)
    []
     [GetSuccess(p2)] -> EXIT )

  ENDPROC (*minitiate*)


  PROCESS ainitiate[OSAP](aa : aid) : EXIT :=

    OSAP !responder(aa) ?p1 : OSP [IsEnableInd(p1) and (aa eq GetAID(p1))];
    EXIT
```

```
ENDPROC (*ainitiate*)


PROCESS mcommunicate[OSAP](aa : aid) : NOEXIT :=

  mnsm[Osap](aa) ||| mosm[Osap](aa)

ENDPROC (* mcommunicate *)

PROCESS acommunicate[Osap](aa : aid) : NOEXIT :=

  mnsa[Osap](aa) ||| mosa[Osap](aa)

ENDPROC (* acommunicate *)


PROCESS mosm[OSAP](aa : aid) : NOEXIT :=

  mpconfirmed[OSAP](initiator(aa))
|||
  munconfirmed[OSAP](initiator(aa))
|||
  (  muserconfirmed[OSAP](initiator(aa))
  || validoperations[OSAP](initiator(aa)))
|||
  i;
  mosm[OSAP](aa)

ENDPROC (* mosm *)


PROCESS mosa[OSAP](aa : aid) : NOEXIT :=

  apconfirmed[OSAP](responder(aa))
|||
  aunconfirmed[OSAP](responder(aa))
|||
  auserconfirmed[OSAP](responder(aa))
|||
  i;
  mosa[OSAP](aa)

ENDPROC (* mosa *)

PROCESS mpconfirmed[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsDisableReq(p1)];
  EXIT

ENDPROC (* mpconfirmed *)

PROCESS apconfirmed[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsDisableReq(p1)];
  EXIT

ENDPROC (* apconfirmed *)


PROCESS munconfirmed[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsUsetReq(p1)];
  EXIT

ENDPROC (* munconfirmed *)


PROCESS aunconfirmed[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsUsetInd(p1)];
  EXIT

ENDPROC (* aunconfirmed *)


PROCESS muserconfirmed[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsReq(p1) and IsUserConf(p1)];
  OSAP !x ?p2 : OSP [p2 ConfForReq p1];
  EXIT

ENDPROC (* muserconfirmed *)


PROCESS auserconfirmed[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsInd(p1) and IsUserConf(p1)];
  OSAP !x ?p2 : OSP [p2 ResForInd p1];
  EXIT

ENDPROC (* auserconfirmed *)


PROCESS validoperations[OSAP](x : AI) : NOEXIT :=

  OSAP !x ?p1 : OSP
  [IsCreate(p1) implies
   not((subj(p1) eq initiator(getaid(p1))) or
       (subj(p1) eq responder(getaid(p1))))];
  validoperations[OSAP](x)

ENDPROC


PROCESS mnsm[OSAP](aa : aid) : NOEXIT :=

  mreport[OSAP](initiator(aa))
|||
  mnotification[OSAP](initiator(aa))
|||
  i;
  mnsm[OSAP](aa)

ENDPROC (* mnsm *)


PROCESS mnsa[OSAP](aa : aid) : NOEXIT :=

  areport[OSAP](responder(aa))
|||
  anotification[OSAP](responder(aa))
|||
  i;
  mnsa[OSAP](aa)

ENDPROC (* mnsa *)


PROCESS mreport[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsReportInd(p1)];
  EXIT

ENDPROC (* mreport *)


PROCESS mnotification[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [Isnotification(p1)];
  EXIT

ENDPROC (* mnotification *)


PROCESS areport[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [IsReportReq(p1)];
  EXIT

ENDPROC (* areport *)


PROCESS anotification[OSAP](x : AI) : EXIT :=

  OSAP !x ?p1 : OSP [Isnotification(p1)];
  EXIT

ENDPROC (* anotification *)
```

```
  PROCESS forcerelease[OSAP](x : AI) : EXIT :=

    OSAP !x ?p1 : OSP [IsDisableConf(p1) or IsDisableInd(p1)];
    EXIT

  ENDPROC (* forcerelease *)


  PROCESS initiaterelease[OSAP](x : AI) : NOEXIT :=

    OSAP !x ?p1 : OSP;
    ([not(IsDisableReq(p1))] -> initiaterelease[OSAP](x)
    []
     [IsDisableReq(p1)] -> releasing[OSAP](x)
    )
  WHERE

    PROCESS releasing[Osap](y : AI) : NOEXIT :=

      OSAP !y ?p1 : OSP [IsConf(p1) or IsDisableInd(p1)];
      releasing[Osap](y)

    ENDPROC (* releasing *)

  ENDPROC (* initiaterelease *)


ENDSPEC (* OM_Protocol *)
```

Appendix C

The Testcases

The testcases specified in this Section run parallel to the service specification, with a restricted set of *all* associations: only two associations are modelled. Each testcase specifies a unique aspect the service specification must exhibit. Note that these testcases are in fact may-testcases.

C.0.1 Testcase No. 1

This testcase deals with the setup of an association, which does *not* succeed, i.e. the manager requests an association but receives a negative confirmation. After that, another attempt is made. Note that, after the enable request, a choice can be made to either receive an enable indication (which would not be according to the testcase) or an enable confirmation. This testcase is described as follows:

```
PROCESS test1[OSAP] : EXIT :=

  (* first attempt *)
  OSAP !offset !enablereq(aid(offset,neighbour(offset)),0);
  OSAP !offset ?p : OSP[IsEnableConf(p) and not(getsuccess(p))];

  (* second attempt *)
  OSAP !offset !enablereq(aid(offset,neighbour(offset)),succ(0));
  OSAP !offset ?p : OSP[IsEnableConf(p) and not(getsuccess(p))];
  EXIT

ENDPROC (* test1 *)
```

Note that the agent does not seem to play a part in this test, however, the decision not to allow the association to be set up could have been made by the agent's service provider.

C.0.2 Testcase No. 2

This testcase deals with the setup of an association, which does succeed, i.e. the manager requests an association and receives a positive confirmation. The next step is to terminate the association, which in this case is invoked by the manager. No data is transmitted in this testcase. This testcase is described as follows:

```
PROCESS test2[OSAP] : EXIT :=

  OSAP
    !offset
    !enablereq(aid(offset,neighbour(offset)),0);
  OSAP
    !offset
    ?p1 : OSP[IsEnableConf(p1) and GetSuccess(p1)];
  OSAP
    !offset
    !disablereq(aid(offset,neighbour(offset)),succ(0));
  OSAP
    !offset
    ?p1 : OSP[IsDisableConf(p1)];
  EXIT
|||
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsEnableInd(p1)];
  OSAP
    !neighbour(offset)
    ?prim : OSP[IsDisableInd(prim)];
  EXIT

ENDPROC (* test2 *)
```

C.0.3 Testcase No. 3

This testcase is similar to the second testcase, however, now instead of the manager, the agent terminates the association. This test is described as follows:

```
PROCESS test3[OSAP] : EXIT :=

  OSAP
    !offset
    !enablereq(aid(offset,neighbour(offset)),0);
  OSAP
    !offset
    ?p1 : OSP[IsEnableConf(p1) and GetSuccess(p1)];
  OSAP
    !offset
    ?prim : OSP[IsDisableInd(prim)];
  EXIT
|||
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsEnableInd(p1)];
  OSAP
    !neighbour(offset)
    !disablereq(aid(offset,neighbour(offset)),succ(0));
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsDisableConf(p1)];
  EXIT

ENDPROC (* test3 *)
```

C.0.4 Testcase No. 4

This testcase deals with the setup of the association, and some messages that use the association. The manager then issues a disable request, which causes some of the confirmations on the messages not to be received. This testcase is described as follows:

```
PROCESS test4[OSAP] : EXIT :=

 (OSAP
    !offset
    !enablereq(aid(offset,neighbour(offset)),0);
  OSAP
    !offset
    ?p1 : OSP[IsEnableConf(p1) and GetSuccess(p1)];
( OSAP
    !offset
    !getreq(aid(offset,neighbour(offset)),succ(0));
  OSAP
    !offset
    ?p1 : OSP[IsGetConf(p1)];
  EXIT
|||
  OSAP
    !offset
    !csetreq(aid(offset,neighbour(offset)),succ(succ(0)));
  EXIT
) >>
  OSAP
    !offset
    !disablereq(aid(offset,neighbour(offset)),succ(succ(succ(0))));
  OSAP
    !offset
    ?p1 : OSP[IsDisableConf(p1)];
  EXIT )
|||
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsEnableInd(p1)];
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsGetInd(p1)];
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsCsetInd(p1)];
  OSAP
    !neighbour(offset)
    !csetres(aid(offset,neighbour(offset)),succ(succ(0)));
  OSAP
    !neighbour(offset)
    !getres(aid(offset,neighbour(offset)),succ(0));
  OSAP
    !neighbour(offset)
    ?prim : OSP[IsDisableInd(prim)];
  EXIT

ENDPROC (* test4 *)
```

C.0.5 Testcase No. 5

This testcase is used to check that if a userconfirmed message is sent, a corresponding confirmation is received. A disable request will be issued after all confirmations have been received.

```
PROCESS test5[OSAP] : EXIT :=

 (OSAP
    !offset
    !enablereq(aid(offset,neighbour(offset)),0);
  OSAP
    !offset
    ?p1 : OSP[IsEnableConf(p1) and GetSuccess(p1)];
( OSAP
    !offset
    !getreq(aid(offset,neighbour(offset)),succ(0));
  EXIT
|||
  OSAP
    !offset
    !csetreq(aid(offset,neighbour(offset)),succ(succ(0)));
  EXIT
) >>
  OSAP
    !offset
    !disablereq(aid(offset,neighbour(offset)),succ(succ(succ(0))));
  OSAP
    !offset
    ?p1 : OSP[IsCsetConf(p1)];
  OSAP
    !offset
    ?p1 : OSP[IsGetConf(p1)];
  OSAP
    !offset
    ?p1 : OSP[IsDisableConf(p1)];
  EXIT )
|||
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsEnableInd(p1)];
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsGetInd(p1)];
  OSAP
    !neighbour(offset)
    ?p1 : OSP[IsCsetInd(p1)];
  OSAP
    !neighbour(offset)
    !getres(aid(offset,neighbour(offset)),succ(0));
  OSAP
    !neighbour(offset)
    !csetres(aid(offset,neighbour(offset)),succ(succ(0)));
  OSAP
    !neighbour(offset)
    ?prim : OSP[IsDisableInd(prim)];
  EXIT

ENDPROC (* test5 *)
```

C.0.6 Testcase No. 6

This testcase is introduced to test the provider initiated disable of the association. First the association is set-up, then some data is exchanged and after that, the Agent decides to terminate the association. However, the Manager receives a provider initiated disable, and so does the Agent, thereby terminating the association.

```
PROCESS test6[OSAP] : EXIT :=

  M[OSAP] ||| A[OSAP]

WHERE

  PROCESS M[OSAP] : EXIT :=

    OSAP
      !offset
      !enablereq(aid(offset,neighbour(offset)),0);
    OSAP
      !offset
      ?p1 : OSP[IsEnableConf(p1) and GetSuccess(p1)];
  ( OSAP
      !offset
      !getreq(aid(offset,neighbour(offset)),succ(0));
    EXIT
  |||
    OSAP
      !offset
      !csetreq(aid(offset,neighbour(offset)),succ(succ(0)));
    EXIT
  ) >>
    OSAP
      !offset
      ?p1 : OSP[IsCsetConf(p1)];
    OSAP
      !offset
      ?p1 : OSP[IsGetConf(p1)];
    (* receive a provider initiated disable            *)
    (* (the id does not relate to the Agent's request) *)
    OSAP
      !offset
      !disableind(aid(offset,neighbour(offset)),succ(succ(succ(succ(succ(0))))));
    EXIT

  ENDPROC (* M *)


  PROCESS A[OSAP] : EXIT :=

    OSAP
      !neighbour(offset)
      ?p1 : OSP[IsEnableInd(p1)];
    OSAP
      !neighbour(offset)
      ?p1 : OSP[IsGetInd(p1)];
    OSAP
      !neighbour(offset)
      ?p1 : OSP[IsCsetInd(p1)];
    OSAP
      !neighbour(offset)
      !getres(aid(offset,neighbour(offset)),succ(0));
    OSAP
      !neighbour(offset)
      !csetres(aid(offset,neighbour(offset)),succ(succ(0)));
    OSAP
      !neighbour(offset)
      !disablereq(aid(offset,neighbour(offset)),succ(succ(succ(0))));
    (* receive a provider initiated disable *)
    OSAP
      !neighbour(offset)
      ?prim : OSP[IsDisableInd(prim)];
    EXIT

  ENDPROC (* A *)

ENDPROC (* test6 *)
```

Bibliography

[AVV97] AVV. *OM/RR and Association Management Specifications*, July 1997.

[BB88] T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS.* Computer Networks and ISDN Systems, 1988.

[BFG$^+$91] A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodriguez, and J.Sifakis. Safety for Branching Time Semantics. In *18th ICALP*. Springer-Verlag, 1991.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[Eer95] H. Eertink. Executing LOTOS specifications: The SMILE tool. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 221–234. Kluwer Academic Publishers, 1995.

[Gar96] Hubert Garavel. An Overview of the EUCALYPTUS Toolbox. Editors Z. Brezočnik and T. Kapus, Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia), University of Maribor, pages 76-88, 1996.

[ISO88] ISO/IEC. LOTOS — *A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* ISO, 1988.

[ISO90a] ISO/IEC. *Common Management Information Protocol Specification.* ISO, 1990.

[ISO90b] ISO/IEC. *Common Management Information Service Definition.* ISO, 1990.

[ISO90c] ISO/IEC. *Remote Operations.* ISO, 1990.

[ISO90d] ISO/IEC. *Systems Management.* ISO, 1990.

[Kap91] Kimberly W. Kappel. *OSI Management Model and Its Impact on Network Management.* Network Management Solutions, Inc, 1991.

[SPKV93] G. Scollo, L. Ferreira Pires, H. Kremer, and C.A. Visser. *Protocol Design.* Twente University, 1993.

[VL86] C.A. Vissers and L. Logrippo. The Importance of the Service Concept in the Design of Data Communications Protocols. *Elsevier Science Publishers B.V. (North-Holland)*, 1986.

[VSvSB91] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science 89, Elsevier*, pages 179–206, 1991.