Linas Laibinis | Elena Troubitsyna |
Sari Leppänen | Johan Lilius | Qaisar A. Malik

Formal Model-Driven Development of Communicating Systems

Turku Centre for Computer Science

TUCS Technical Report
No 691, June 2005

Formal Model-Driven Development of Communicating Systems

Linas Laibinis
Åbo Akademi University, Department of Computer Science
Elena Troubitsyna
Åbo Akademi University, Department of Computer Science
Sari Leppänen
Nokia Research Center, Mobile Networks Laboratory
Johan Lilius
Åbo Akademi University, Department of Computer Science
Qaisar A. Malik
Åbo Akademi University, Department of Computer Science

TUCS Technical Report
No 691, June 2005

Abstract

Telecommunicating systems should have a high degree of availability, i.e., high probability of correct and timely provision of requested services. To achieve this, correctness of software for such systems should be ensured. An application of formal methods helps us to gain confidence in building correct software. However, to be used in practice, the formal methods should be well integrated into existing development process. In this paper we propose a formal model-driven approach to development of communicating systems. Essentially our approach formalizes Lyra – a top-down service-oriented method for development of communicating systems. Lyra is based on transformation and decomposition of models expressed in UML2. We formalize Lyra in the B Method by proposing a set of formal specification and refinement patterns reflecting the essential models and transformations of Lyra. The proposed approach is illustrated by a case study – development of the 3GPP positioning system.

Keywords: Lyra, B Method, 3GPP, Formal Methods, Model-Driven, UML

TUCS Laboratories
Distributed Systems Design laboratory
Embedded Systems laboratory

1. Introduction

Modern telecommunicating systems are usually distributed software-intensive systems providing a large variety of services to their users. Development of software for such systems is inherently complex and error prone. However, software failures might lead to unavailability or incorrect provision of system services which could incur significant financial losses. Hence it is important to guarantee correctness of software for telecommunicating systems.

Formal methods have been traditionally used for reasoning about software correctness. However they are yet insufficiently well integrated into current development practice. Unlike formal methods, Unified Modelling Language (UML) [9] has a lower degree of rigor for reasoning about software correctness but is widely accepted in industry. UML is a general purpose modelling language and, to be used effectively, should be tailored to the specific application domain.

Nokia Research Centre has developed a design method Lyra [7] – a UML-based service-oriented method specific to the domain of communicating systems and communication protocols. The design flow of Lyra is based on concepts of decomposition and preservation of the externally observable behaviour. The system behaviour is modularised and organized into hierarchical layers according to the external communication and related interfaces. It allows the designers to derive the distributed network architecture from the functional system requirements via a number of model transformations. This approach coincides with the stepwise refinement paradigm adopted in the B Method [1].

In this paper we propose a set of formal specification and refinement patterns reflecting the essential models and transformations of Lyra. Our approach is based on stepwise refinement of a formal system model in the B Method [1,12] – a formal framework with an automatic tool support. While developing a system by refinement, we start from an abstract specification and gradually incorporate implementation details into it until an executable code is obtained. While formalizing Lyra, we single out a generic concept of a communicating service component and propose patterns for specifying and refining it. In the refinement process the service component is decomposed into a set of service components of smaller granularity specified according to the proposed pattern. Moreover, we demonstrate that the process of distributing service components between different network elements can also be captured by the notion of refinement. The proposed formal specification and development patterns establish a background for automatic generation of formal specifications from UML models and expressing model transformations as refinement steps. Via automation of the UML-based Lyra design flow we aim at smooth incorporation of formal methods into existing development practice. The proposed approach is illustrated by a case study – development of the 3GPP positioning system.

2. Lyra: Service-Based Development of Communicating Systems

Overview of Lyra. Lyra [7] is a model-driven and component-based design method for the development of communicating systems and communication protocols. It has been developed in Nokia Research Center by integrating the best practices and design patterns established in the area of communicating systems. The method covers all industrial specification and design phases from prestandartization to final implementation. It has been successfully applied in large-scale UML2-based industrial software development.

Lyra has four main phases: Service Specification, Service Decomposition, Service Distribution and Service Implementation. The *Service Specification* phase focuses on defining services provided by the system and their users. The goal of this phase is to define the externally observable behaviour of the system level services via deriving logical user interfaces. In the *Service Decomposition* phase the abstract model produced at the previous stage is decomposed in a stepwise and top-down fashion into a set of service components and logical interfaces between them. The result of this phase is the logical architecture of the service implementations. In *Service Distribution* phase, the logical architecture of services is distributed over a given platform architecture. Finally, in *Service Implementation* phase the structural elements are adjusted and integrated to the target environment, low-level implementation details are added and platform-specific code is generated. Next we discuss Lyra in more detail with an example.

Lyra by example. We model part of a Third Generation Partnership Project (3GPP) positioning system [14,15]. The positioning system provides positioning services to calculate the physical location of a given user equipment (UE) in a Universal Mobile Telecommunication System (UMTS) network. We focus on Position Calculation Application Part (PCAP) – a part of the positioning system allowing communication in the Radio Access Network (RAN). PCAP manages the communication between the Radio Network Controller (RNC) and the Stand-alone Assisted Global Positioning System Serving Mobile Location Centre (SAS) network elements. The functional requirements for the RNC-SAS communication have been specified in [14, 15].

The Service Specification phase starts from creating a domain model of the system. It describes the system with the included system level services and different types of external users. Each association connecting an external user and a system level service corresponds to a logical interface. For the system and the system level services we define active classes, while for each type of an external user we define the corresponding external class. The relationships between the system level services and their users become candidates *for PSAPs – Provided Service Access Points* of the system level services. The logical interfaces are attached to the classes with ports. The domain model for the *Positioning system* and its service *PositionCalculation* is shown in Fig.1a and PSAP of the Positioning system – *I_User PSAP* is shown in Fig.1b. The UML2 interfaces *I_ToPositioning* and *I_FromPostioning* define the signals and signal parameters of *I_user PSAP*.

A valid execution order of signals on PSAP can be specified by the corresponding use case and sequence diagrams. For the *Positioning* system, the use case diagram

would merely depict splitting the *PositionCalculation* use case into two main use cases: successful and unsuccessful. The sequence diagrams would draft the communication in each use case. (We omit the presentation of diagrams for brevity). Finally, we formally describe the communication between a system level service and its user(s) in the *PSAPCommunication* state machine as illustrated in Fig.1c. The positioning request *pc_req* received from the user is always replied: with the signal *pc_cnf* in case of success, and with the signal *pc_fail_cnf* otherwise.

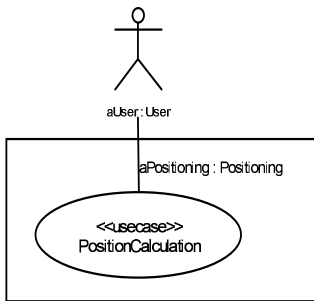


Fig.1a. Domain model

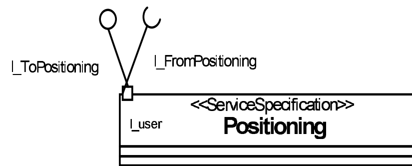


Fig.1b. PSAP of Positioning System

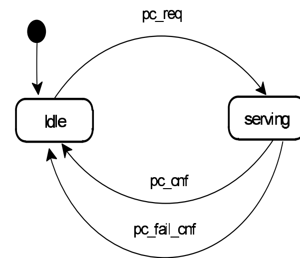


Fig.1c. State diagram of PSAP Communication

To implement its own services, the system usually uses external entities. For instance, to provide the *PositionCalculation* service, the positioning system should first request Radio Network Database *(DB)* for an approximate position of User Equipment (*UE)*. The information obtained from *DB* is used to contact *UE* and request it to emit a radio signal. At the same time, the Reference Local Measurement Unit *(ReferenceLMU)* is requested to emit a radio signal. The strengths of radio signals obtained from *UE* and *ReferenceLMU* are used to calculate the exact position of *UE*. The calculation is done by Algorithm service provider (*Algorithm)* which finally provides the user with the final estimation of the UE location. Let us observe that services provided by the external entities partition execution of the *PositionCalculation* service into the corresponding stages. In the next phase of Lyra development – *Service Decomposition* – we focus on specifying service execution according to the identified stages.

In this phase we introduce external service providers into the domain model constructed previously, as shown in Fig.2a. The model includes the external service providers *DB, UE, ReferenceLMU* and *Algorithm* which are then defined as external classes. For each association between a system level service and an external class we define a logical interface. The logical interfaces are attached to the corresponding classes via ports called *USAPs – Used Sevice Access Points* as presented in Fig.2b.

To specify the required stages of service implementation, we decompose the behaviour of the main use cases accordingly. For instance, the successful calculation of UE position can be decomposed as shown in Fig.2c. The sequence diagrams (omitted here) are created to model the signalling scenarios for each stage of service implementation. Observe that the behaviour is modularised according to the related service access points – PSAPs and USAPs. Moreover, the functional architecture is

defined in terms of service components, which encapsulate functionalities related to a single execution stage or other logical piece of functionality.

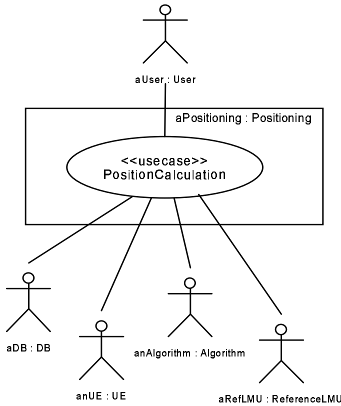


Fig. 2a. Domain model

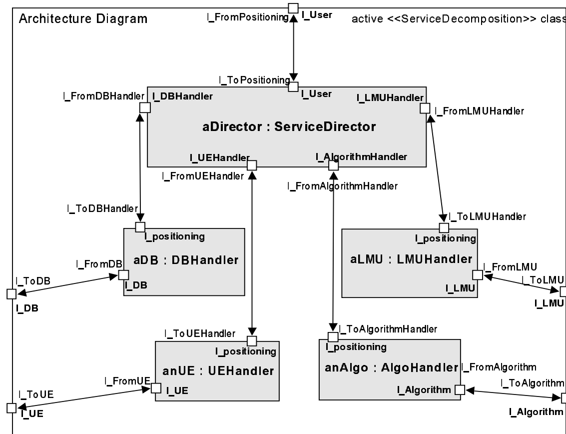


Fig. 2d. PositionCalculation functional architecture

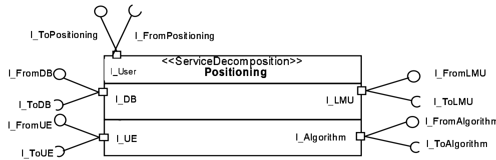


Fig. 2b. PSAP and USAPs of Positioning

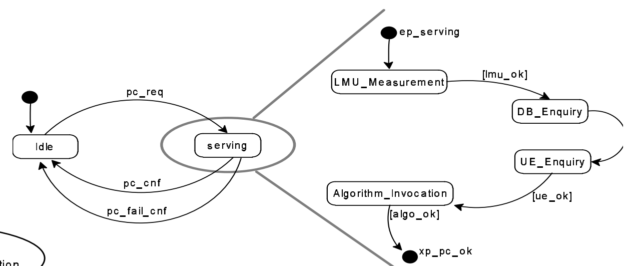


Fig. 2e. ServiceDirector: PSAP communication and execution control

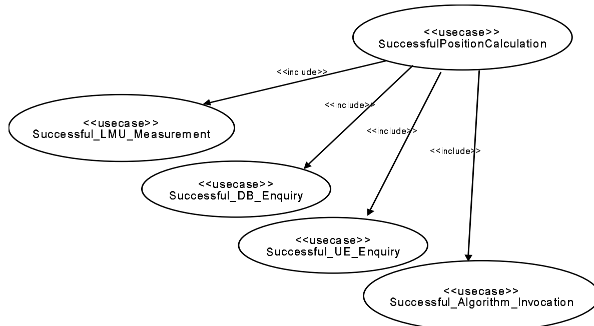


Fig. 2c. Use case decomposition

In Fig.2d we present the architecture diagram of the *Positioning* system. *ServiceDirector* plays two roles: it manages the execution control in the system and handles the communication on the PSAP. The behaviour of *ServiceDirector* is presented in Fig.2e. The top-most state machine specifies the communication on PSAP, while the submachine state *Serving* specifies a valid execution flow of the position calculation. The substates of *Serving* encapsulate the stage-specific behaviour and can be represented as the corresponding submachines. These machines (omitted here) in their turn include specifications of the PSAP-USAP communication.

The modular system model produced at the Service Decomposition phase allows us to analyse various distribution models. In the next phase – Service Distribution – the service components are distributed over a given network architecture. Signalling protocols allow for communication between the service components in distant network elements.

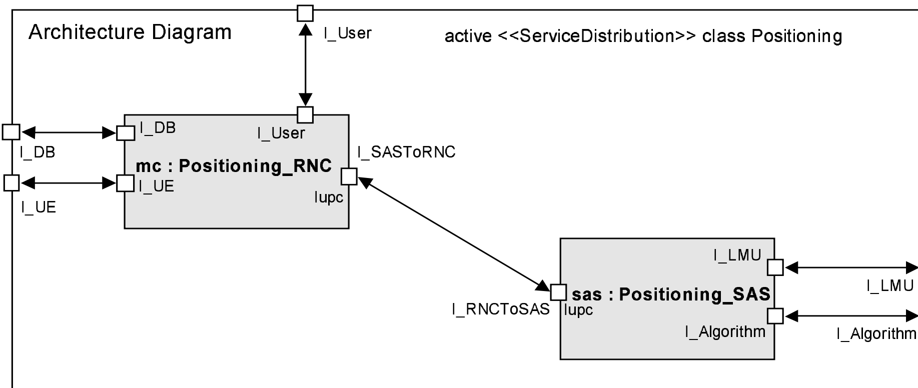


Fig. 3a. Architecture of service distribution

In Fig.3a we illustrate the physical structure of the distributed positioning system. *Positioning_RND* and *Positioning_SAS* represent network elements in a UMTS network. Protocol Data Unit (PDU) interface *Iupc* is used in communicating between the network elements. We map the functional architecture to the physical structure by including the service components into the network elements. The functional architecture of the SAS network element is illustrated in Fig.3b. The functionality of *ServiceDirector* specified at the Service Decomposition phase is now decomposed and distributed over the given network. *ServiceDirector_SAS* handles the PDU interface towards RNC network element and controls the execution flow of the positioning calculation process in the SAS network element.

Finally, at the *Service Implementation* phase we specify how the virtual PDU communication between entities in different network nodes is realized using the underlying transport services. We also implement data encoding and decoding, routing of messages and dynamic process management. The detailed discussion of this stage can be found elsewhere [7, 14, 15].

In the next section we give a brief introduction into our formal framework – the B Method, which we will use to formalize the development flow described above.

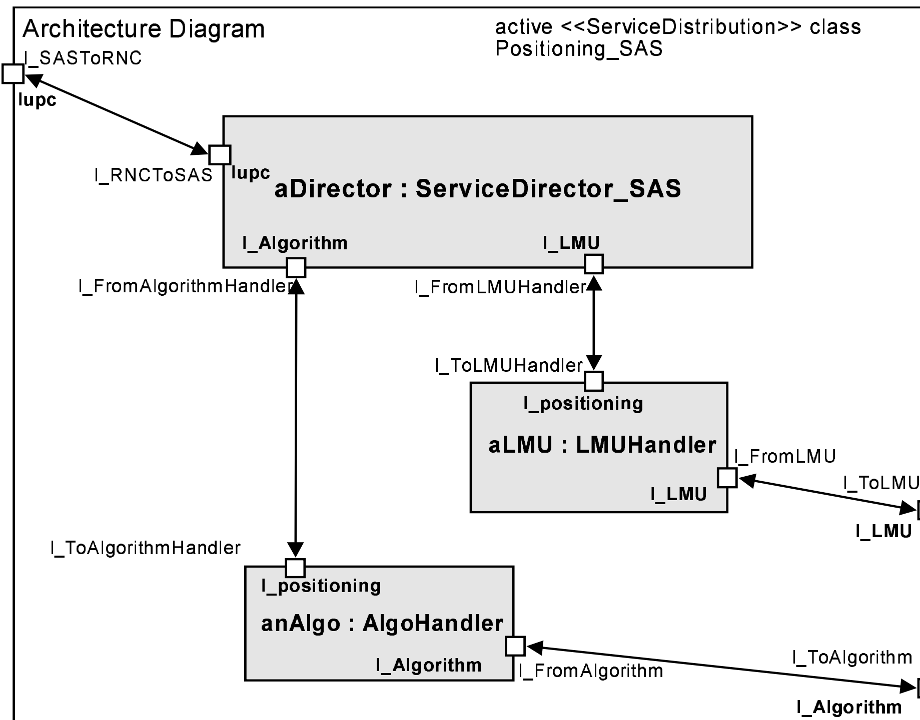


Fig. 3b. Architecture of Positioning_SAS

3. Modelling in the B Method

The B Method: background. The B Method [1] (further referred to as B) is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [4,8]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [12], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement [1]. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called *refinements*. A formal specification is a mathematical model of the required behaviour of a (part of) system. In B a specification is represented by a set of modules, called Abstract Machines. An abstract machine encapsulates state and operations of the specification and as a concept is similar to a module or a package.

Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialised in the INITIALISATION clause.

The variables in B are strongly typed by constraining predicates of the INVARIANT clause. All types in B are represented by non-empty sets.

The operations of the machine are defined in the OPERATIONS clause. In this paper we use Event B extension of the B Method. The operations in Event B are described as guarded statements of the form SELECT cond THEN body END. Here cond is a state predicate, and body is a B statement. If cond is satisfied, the behaviour of the guarded operations corresponds to the execution of their bodies. However, if cond is false, then the execution of the corresponding operation is suspended, i.e., the operation is in waiting mode until cond becomes true.

B statements that we are using to describe a state change in operations have the following syntax:

```
S == x := e | IF cond THEN S1 ELSE S2 END | S1 ; S2 | x :: T |
     S1 || S2 | ANY z WHERE cond THEN S END | ...
```

The first three constructs – assignment, conditional statement and sequential composition (used only in refinements) have the standard meaning. The remaining constructs allow us to model nondeterministic or parallel behaviour in a specification. Usually they are not implementable so they have to be refined (replaced) with executable constructs at some point of program development. The detailed description of the B statements can be found elsewhere [1].

The B method provides us with mechanisms for structuring the system architecture by modularisation. A module is described as a machine. The modules can be composed by means of several mechanisms providing different forms of encapsulation. For instance, if the machine C INCLUDES the machine D then all variables and operations of D are visible in C. However, to guarantee internal consistency (and hence independent verification and reuse) of D, the machine C can change the variables of D only via the operations of D. In addition, the invariant properties of D are included into the invariant of C.

To illustrate basic principles of specifying and refining in B, next we present our approach to formal specification of a service component.

Modelling Service Component in B. Let us remind that we have described a service component as a coherent piece of functionality which provides its services to a service consumer via PSAP. We used this term to refer to external service providers introduced at the Service Decomposition phase. However, the notion of a service component can be generalized to represent service providers at the different levels of abstraction. Indeed, even the entire *Positioning* system can be seen as the service component providing the *Position Calculation* service. On the other hand, peer proxies introduced at the lowest level of abstraction can also be seen as the service components providing the physical data transfer services. Therefore, the notion of a service component is central to the entire Lyra development process.

A service component has two essential parts: functional and communicational. The functional part is a "mission" of a service component, i.e., the service(s) which it is capable of executing. The communicational part is an interface via which the service

component receives requests to execute the service and sends the results of service execution.

```
MACHINE  ACC
VARIABLES  inp_chan, input, out_chan, output
INVARIANT
  inp_chan : INPUT_DATA & input : INPUT_DATA &
 out_chan : OUT_DATA &   output : OUT_DATA
INITIALISATION
  inp_chan, input := INPUT_NIL, INPUT_NIL ||
  out_chan, output := OUT_NIL, OUT_NIL
OPERATIONS
```

```
ACM
env_req =
  SELECT inp_chan = INPUT_NIL
  THEN
    inp_chan :: INPUT_DATA - {INPUT_NIL}
  END;
read =
  SELECT not(inp_chan = INPUT_NIL) &
            (input = INPUT_NIL)
  THEN
    input,inp_chan := inp_chan,INPUT_NIL
  END;
write =
  SELECT not(output = OUT_NIL) &
            (out_chan = OUT_NIL)
  THEN
    out_chan,output := output,OUT_NIL
  END;
env_read =
  SELECT not(out_chan = OUT_NIL)
  THEN
    out_chan := OUT_NIL
  END
```

```
ACAM

calculate =
  SELECT not(input = INPUT_NIL) &
            (output = OUT_NIL)
  THEN
    CHOICE
      output ::
          OUT_DATA - {OUT_NIL,OUT_FAIL}
    OR
      output := OUT_FAIL
    END ||
    input := INPUT_NIL
  END;

END
```

Usually execution of a service involves certain computations. We call the B representation of this part of service component an *Abstract CAlculating Machine (ACAM)*. The communicational part is correspondingly called *Abstract Communicating Machine (ACM)*, while the entire B model of a service component is called *Abstract Communicating Component (ACC)*. The abstract machine ACC below presents the proposed pattern for specifying a service component in B.

In our specification we abstract away from the details of computations required to execute the service. Our specification of *ACAM* is merely a statement non-deterministically generating results of service execution in case of success or failure. The communication with a service component is conducted via two channels – inp_chan and out_chan – shared between the service component and the service consumer. While specifying a service component, we adopt a systems approach, i.e., model the service component together with the relevant part of its environment, the service consumer. Namely, we model how the service consumer places requests to

execute a service in the operation env_req and reads the results of service execution in the operation env_resp.

The operations read and write are internal to the service component. The service component reads the requests to execute a service from inp_chan as defined in the operation read. As a result of read execution, the request is stored into the internal data buffer input, so it can be used by *ACAM* while performing the required computing. Symmetrically the operation write models placing the results of computations performed by *ACAM* into the output channel, so it can be read by the service consumer. We reserve the abstract constant NIL to model the absence of data, i.e., the empty channels. The operations discussed above model the *ACM* part of *ACC*.

We argue that the machine ACC can be seen as a specification pattern which can be instantiated by supplying the details specific to a service component under construction. For instance, the *ACM* part of *ACC* models data transfer to and from the service component very abstractly. While developing a realistic service component, this part can be instantiated with real data structures and corresponding protocols for transferring them.

In the next section we demonstrate how Lyra development flow can be formalized as refinement and decomposition of ACC.

4. Formal Service-Oriented Development

As described in Section 2, usually a service component is represented as an active class with the PSAP attached to it via the port. The state diagram depicts signalling scenario on PSAP including the signals from and to the external class modelling the service consumer. Essentially these diagrams suffice to specify the service component according to the pattern *ACC* proposed in Section 3. The general principle of translation is shown in Fig.4.

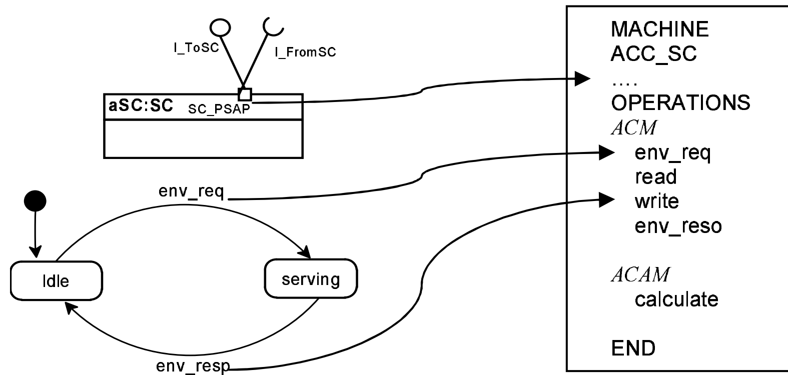


Fig.4. Translating UML2 model into the ACC pattern

The UML2 description of PSAP of the service component *SC* is translated into the *ACM* part of the machine ACC_SC specifying *SC* according to the *ACC* pattern. The *ACAM* part of ACC_SC instantiates the non-deterministic assignment of ACC by the

data types specific to the modelled service component. These translations formalize the *Service Specification* phase of Lyra.

In the next phase of Lyra development – *Service Decomposition* – we decompose the service provided by the service component into a number of stages (subservices). The service component can execute certain subservices itself as well as request the external service components to do it. At the *Service Decomposition* phase two major transformations are performed:

- the service execution is decomposed into a number of stages (or subservices), and
- communication with the external entities executing these subservices is introduced via USAPs.

Each transformation corresponds to a separate refinement step in our approach.

According to Lyra, the flow of the service execution is orchestrated by *Service Director* (often called a Mediator). It implements the behaviour of PSAP of the service component as specified earlier, as well as co-ordinates execution by enquiring the required subservices from the external entities according to the defined execution flow.

Assume that the service component *SC* specified by the machine ACC_SC at the Service Specification phase is providing the service *S* which is decomposed into the subservices *S1, S2*, and *S3*. Moreover, let assume that the state machine of *Service Director* defines the desired order of execution: first *S1*, then *S2* and finally *S3*. The UML2 representation of this is given in Fig.5, in which we also demonstrate that in B such decomposition can be represented as a refinement of our abstract pattern *ACC* instantiated to model *SC*.

This step focuses on refinement of the *ACAM* part of *ACC*. As in *ACAM*, in the refinement of it - *ACAM'*- the operation calculate puts the results of service execution on the output channel. However, calculate is now preceded by the operation director, which models *Service Director* orchestrating the stages of execution. We introduce the variables S1_data, S2_data and S3_data to model the results of execution of the corresponding stages. The operation director specifies the desired execution flow by assigning corresponding values to the variable curr_service. In general, execution of any stage of service can fail. In its turn, this might lead to failure of the entire service provision. In the Appendix you can find the detailed B development of the positioning system described in Section 2. While specifying *Service Director,* we abstractly model error recovery – upon detecting an error, *Service Director* can retry (up to the predefined number of attempts) to execute a certain stage of the service. However, if error recovery fails, the service director terminates the service execution and returns the error as the final result. The error detection is abstractly modelled by using special evaluation functions which classify the results of the corresponding service stages into three categories: success, a recoverable error, an unrecoverable error.

Unlike in Lyra, in our B development the *Service Decomposition* and *Service Distribution* phases are not entirely disjoint. This is explained by the fact that the INCLUDES structuring mechanism enforces the master-slave relationship between components, i.e., the including machine has complete control over the included machine. As a result, modelling of communication between two peer components is cumbersome. However, this problem can be alleviated if the targeted service distribution is taken into account while introducing the communication with the external service components via USAPs.

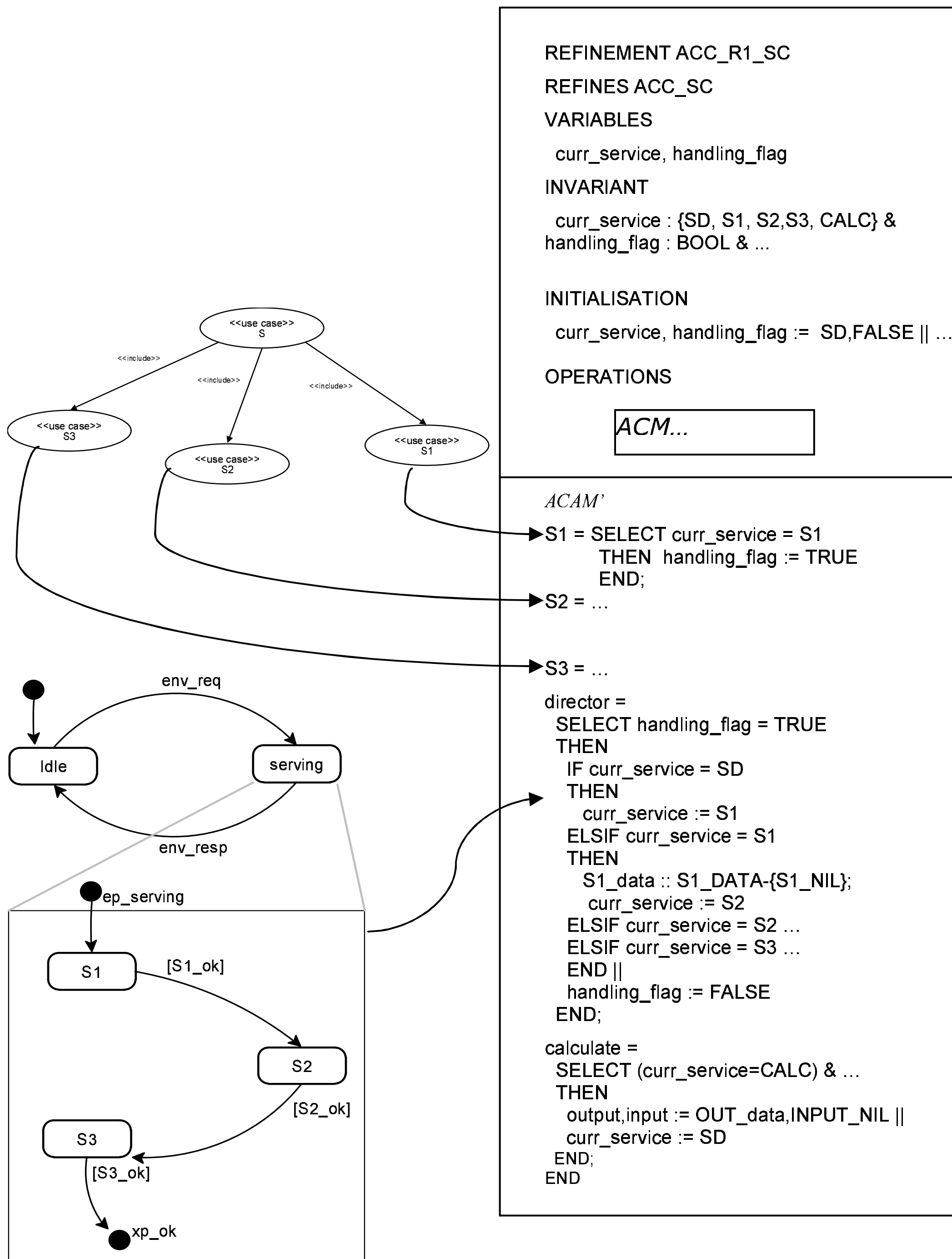


Fig.5. Service decomposition and refinement

To derive the pattern for translating UML2 diagrams modelling functional and platform distributed service architecture at these two phases we should consider two general cases:

1) the service director of *SC* is "centralized", i.e., it resides on a single network element,
2) the service director of *SC* is "distributed", i.e., different parts of execution flow are orchestrated by distinct service directors residing on different network elements. The service directors communicate with each other while passing the control over the corresponding parts of the flow.

In both cases the model of the service component SC with USAPs looks as shown in Fig.6. The service distribution architecture diagram for the first case is given in Fig.7.

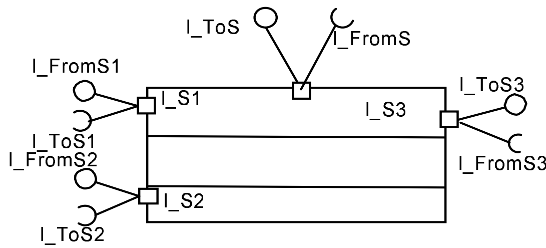


Fig.6. Service component with USAPs

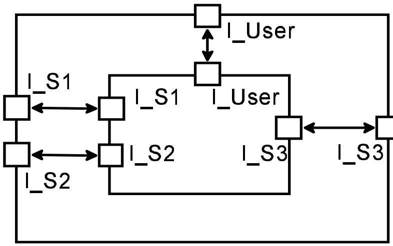


Fig.7. Architecture diagram (case 1)

It is easy to observe that the service component *SC* plays a role of the service consumer for the service components *SC1, SC2* and *SC3*. We specify the service components *SC1, SC2* and *SC3* as separate machines ACC_SC1, ACC_SC2, ACC_SC3 according to the proposed pattern *ACC* as depicted in Fig.8. The process of translating their UML2 models into B is similar to specifying *SC* at the *Service Specification* phase. The *ACM* parts of the included machines specify their PSAPs. To ensure the match between the corresponding USAPs of *SC* and PSAPs of the external service components, we derive USAPs of *SC* from PSAPs of *SC1, SC2* and *SC3*.

Besides defining separate machines to model external service components, in this refinement step we also define the mechanisms for communicating with them. We refine the operation director to specify communication on USAPs. Namely, we replace non-deterministic assignments modelling stages of service execution by the corresponding signalling scenario: at the proper point of the execution flow director requests a desired service by writing into the input channel of the corresponding included machine, e.g., SC1_write_ichan, and later reads the produced results from the output channel of this machine, e.g., SC1_read_ochan. Graphically this arrangement is depicted in Fig.9.

Modelling the case of the distributed service director is more complex. Let assume that the execution flow of the service component *SC* is orchestrated by two service directors: the *ServiceDirector1,* which handles the communication on PSAP of *SC* and communicates with *SC1,* and *ServiceDirector2,* which orchestrates the execution of *S2* and *S3*. The architecture diagram depicting the overall arrangement is shown in Fig.10.

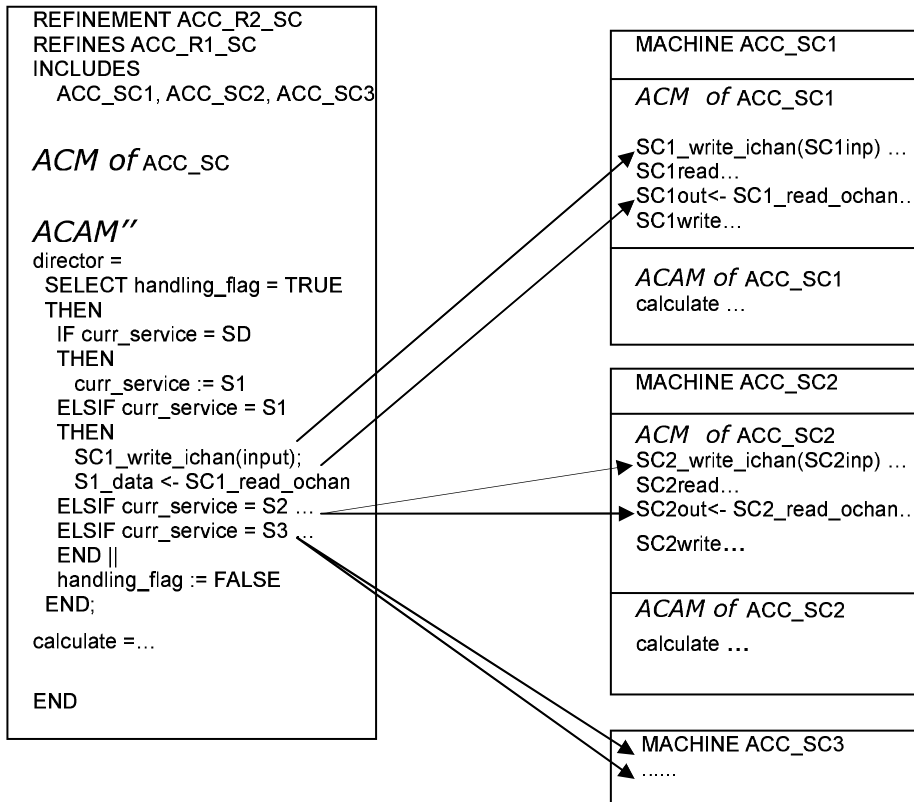


Fig.8. Refinement at *Service Decomposition* and *Service Distribution* phases

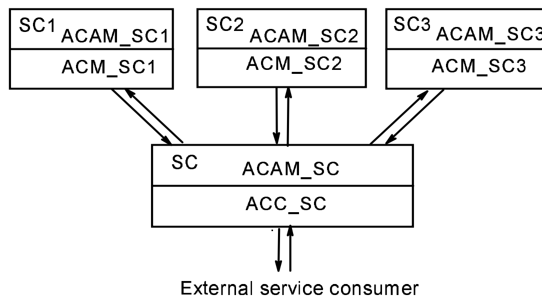


Fig.9. Architecture of formal specification

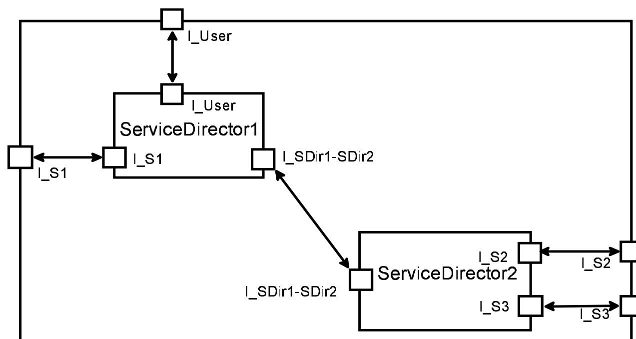


Fig.10. Architecture diagram (case 2)

The service execution proceeds according to the following scenario: via PSAP of *SC ServiceDirector1* receives the request to provide the service *S*. Upon this, via USAP of *SC,* it requests the component *SC1* to provide the service *S2*. After the result of *S2* is obtained, *ServiceDirector1* requests *Service Director2* to execute the rest of the service and return the result back. In its turn, *ServiceDirector2* at first requests *SC2* to provide the service S2 and then *SC3* to provide service *S3*. Upon receiving the result from *S3,* it forwards it to *ServiceDirector1*. Finally, *Service Director1* returns to the service consumer the result of the entire service *S* via PSAP of *SC*.

This complex behaviour can be captured in a number of refinement steps. At first, we observe that *ServiceDirector2* co-ordinating execution of *S2* and *S3* can be modelled as a "large" service component *SC2-SC3* which provides the services *S2* and *S3*. Let us note that the execution flow in *SC2-SC3* is orchestrated by a "centralized" service director *ServiceDirector2.* We use this observation in our next refinement step. Namely we refine the B machine modelling *SC* by including into it the machines modelling the service components *SC1* and *SC2-SC3* and introducing the required communicating mechanisms. In our consequent refinement step we focus on decomposition of *SC2-SC3*. The decomposition is performed according to the proposed scheme: we introduce the specification of *ServiceDirector2* and decompose *ACAM* of *SC2-SC3*. Finally, we single out separate service components *SC2* and *SC3* as before and refine *ServiceDirector2* to model communication with them. The final architecture of formal specification is shown in Fig.11. We omit the presentation of the detailed formal specifications – they are again obtained by the recursive application of the proposed specification and refinement patterns. The full B specifications (specialized for the positioning system) can be found in the Appendix.

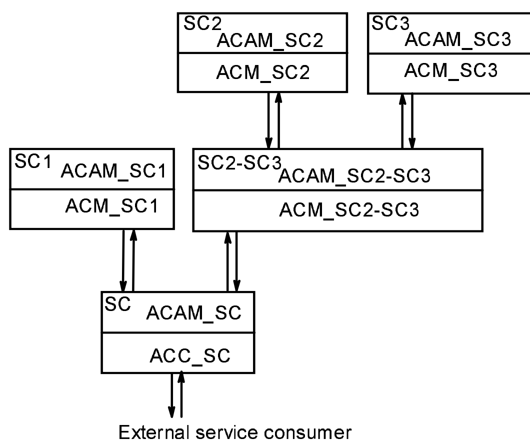


Fig.11. Architecture of formal specification (case 2)

At the consequent refinement steps we focus on particular service components and refine them (in the way described above) until the desired level of granularity is obtained. Once all external service components are in place, we can further decompose their specifications by separating their *ACM* and *ACAM* parts. Such decomposition will allow us to concentrate on the communicational parts of the respective components and

further refine them by introducing details of required concrete communication protocols.

Discussion. While describing formalisation of Lyra in B, we considered the sequential model of service execution. However, a parallel execution of services is also a valid interpretation of the considered UML2 models. In event-based B development, which we used, parallelism is modelled via the interleaving semantics. Observe that if some operations are enabled simultaneously, they can be executed in any order or in parallel provided they do not have a conflict on the variables.

Though not presented in this paper, we have succeeded in modelling parallel execution starting from the *Service Decomposition* phase. At the *Service Distribution* phase, in case of a "centralised" service director the parallelism is preserved. However, in case of a "distributed" service director preserving parallelism might require additional communication between the service directors or a part of parallelism might be lost.

5. Conclusions

In this paper we proposed a formal approach to development of communicating distributed systems. Our approach formalizes Lyra [7] – the UML2-based design methodology adopted in Nokia. The formalization is done within the B Method [1,12] – the formal framework supporting system development by stepwise refinement. We derived the B specification and refinement patterns reflecting models and model transformations used in the development flow of Lyra. The proposed approach establishes a basis for automatic translation of UML2-based development of communicating systems into the specification and refinement process in B. Hence UML2 modelling can be seen as a syntactic sugaring of the formal development. However, such syntactic sugaring enables a smooth integration of formal methods into existing development practice. Since UML is widely accepted in industry we believe that our approach has a potential for wide industrial uptake.

Lyra adopts the service-oriented style for development of communicating systems. We presented the guidelines for deriving B specifications from corresponding UML2 models at each development stage of Lyra and validated the development by the corresponding B refinements. The major model transformations aim at service decomposition and distribution over the given platform. The proposed formal model of communication between the distributed service components is generic and can be instantiated by virtually any concrete communication protocol.

The initial formalization of Lyra has been undertaken using model checking techniques [7]. However, because telecommunicating systems tend to be large and data intensive this formalization was prone to the state explosion problem. Our approach helps to overcome this limitation.

Development of distributed communicating systems has been a topic of ongoing research over several decades. Our review of related work is confined by the consideration of the recent research conducted within the B Method.

Treharne et all. [13] investigated verification of safety and liveness properties of communicating components by combining the B Method and the process algebra CSP. However, they do not consider service decomposition and distribution aspects of communicating system development.

Boström and Walden [2] proposed a formal methodology (based on the B Method) for developing distributed grid systems. In their approach the B language is extended with grid-specific features. In their work, the system development is governed by B refinement. In our approach the system development is guided by the existing development practice, so that the refinement process is hidden behind the facade of UML.

There is an active research going on translating UML to B [3,5,6,10,11]. Among these, the most notable is research conducted by Snook and Butler [10] on designing the method and the U2B tool to support the automatic translation. In our future work we are planning to integrate our efforts with Snook and Butler to achieve the automatic translation of Lyra into B. While doing this, we will focus specifically on translating models and model transformations used in Lyra to automate formalisation of the entire UML-based development process in the domain of the communicating distributed systems. Furthermore, we are planning to further enhance the proposed approach to address issues of fault tolerance, concurrency and integration of process algebraic approaches to verify the dynamic properties of communication protocols between network elements.

Acknowlegements

This work is supported by EU funded research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] P.Boström and M.Waldén. *An Extension of Event B for Developing Grid Systems*, in Helen Treharne, Steve King, Martin Henson (Eds.), Proceedings of Formal Specification and Development in Z and B: 4th International Conference, Guildford, UK, April 13-15, 2005.

[3] P.Facon, R.Laleau, H.P.Nguyean, and A.Mammar. Combining UML with the B formal method for the specification of database applications. *Research report, CEDRIC laboratory*, Paris, 1999.

[4] L.Laibinis and E.Troubitsyna. *Fault Tolerance in a Layered Architecture: A General Specification Pattern in B.* Proceedings of 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China, September 2004. IEEE Press, pp.346-355.

[5] K.Lano, D.Clark, and K.Adroutsopoulos. UML to B: Formal Verification of Object-Oriented Models. In E.A.Boiten, J.Derrick, G.Smith (Eds.): *Integrated Formal Methods*, *4th International Conference*, IFM 2004. Springer, LNCS 2999, pp. 187-206.

[6] H.LeDang and J.Souquieres. Integrating UML and B specification techniques. In proceedings of *Informatik2001 Workshop on Integrating Diagrammatic and Formal Specification Techniques*, 2001.

[7] S.Leppänen, M.Turunen, and I.Oliver. *Application Driven Methodology for Development of Communicating Systems*. FDL'04, Forum on Specification and Design Languages. Lille, France, September 2004.

[8] MATISSE Handbook for Correct Systems Construction. 2003. http://www.esil.univ-mrs.fr/~spc/matisse/Handbook/

[9] J.Rumbaugh, I.Jacobson, and G.Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1998.

[10] C.Snook and M.Butler. *U2B - A tool for translating UML-B models into B*, in Mermet, J., Eds. *UML-B Specification for Proven Embedded Systems Design*, chapter 6. Springer, 2004.

[11] C.Snook and M.Waldén. *Use of U2B for Specifying B Action Systems* (Extended abstract). In Proceedings of RCS'02 - International workshop on Refinement of Critical Systems: Methods, Tools and Experience, Grenoble, France, January 2002.

[12] Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 2001. Available at http://www.atelierb.societe.com/index uk.html

[13] H.Treharne, S.Schneider, and M.Bramble. Composing Specifications Using Communication, in D. Bert, J.P. Bowen, S. King, M. Waldén (Eds.), Proceedings of Formal Specification and Development in Z and B: 3rd International Conference, Turku, Finland, June 4-6, 2003.

[14] 3GPP. Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. See http://www.3gpp.org/ftp/Specs/html-info/25305.htm

[15] 3GPP. Technical specification 25.453: UTRAN Iupc interface positioning calculation application part (pcap) signalling. See http://www.3gpp.org/ftp/Specs/html-info/25453.htm

Appendix

```
MACHINE
  Main

SEES
  Comp_data

VARIABLES
  inp_chan, input, out_chan, output

INVARIANT
  inp_chan : INPUT_DATA &
  input : INPUT_DATA &
  out_chan : POS_DATA &
  output : POS_DATA

INITIALISATION
  inp_chan, input := INPUT_NIL, INPUT_NIL ||
  out_chan, output := POS_NIL, POS_NIL

OPERATIONS

env_write =
  SELECT inp_chan = INPUT_NIL
  THEN
    inp_chan :: INPUT_DATA - {INPUT_NIL}
  END;

read =
  SELECT not(inp_chan = INPUT_NIL) & (input = INPUT_NIL)
  THEN
    input,inp_chan := inp_chan,INPUT_NIL
  END;

db =
  SELECT not(input = INPUT_NIL)
  THEN
    skip
  END;

ue =
  SELECT not(input = INPUT_NIL)
  THEN
    skip
  END;

lmu =
  SELECT not(input = INPUT_NIL)
  THEN
    skip
  END;

pos =
```

```
  SELECT not(input = INPUT_NIL)
  THEN
    skip
  END;

handle =
  SELECT not(input = INPUT_NIL)
  THEN
    skip
  END;

calculate =
  SELECT not(input = INPUT_NIL) & (output = POS_NIL)
  THEN
    output :: POS_DATA - {POS_NIL} ||
    input := INPUT_NIL
  END;

write =
  SELECT not(output = POS_NIL) & (out_chan = POS_NIL)
  THEN
    out_chan,output := output,POS_NIL
  END;

env_read =
  SELECT not(out_chan = POS_NIL)
  THEN
    out_chan := POS_NIL
  END

END
```

```
MACHINE Comp_data

SETS
 POS_DATA; INPUT_DATA; DB_DATA; UE_DATA; LMU_DATA;
 STATUS = {OK,RECOV,UNRECOV}

CONSTANTS
 POS_NIL, POS_FAIL, INPUT_NIL, DB_FAIL, UE_FAIL, LMU_FAIL,
 DB_NIL, UE_NIL, LMU_NIL, LMU_OK,
 DB_Eval, UE_Eval, LMU_Eval, POS_Eval,
 N_DB, N_UE, N_LMU, N_POS

PROPERTIES
  POS_NIL : POS_DATA & POS_FAIL : POS_DATA & not(POS_NIL = POS_FAIL) &
  INPUT_NIL : INPUT_DATA &
  DB_FAIL : DB_DATA & DB_NIL : DB_DATA & not(DB_FAIL = DB_NIL) &
  UE_FAIL : UE_DATA & UE_NIL : UE_DATA & not(UE_FAIL = UE_NIL) &
  LMU_FAIL : LMU_DATA & LMU_NIL : LMU_DATA & not(LMU_FAIL = LMU_NIL) &
  LMU_OK : LMU_DATA & not(LMU_OK = LMU_NIL) & not(LMU_OK = LMU_FAIL) &
  DB_Eval: DB_DATA --> STATUS &
  UE_Eval: UE_DATA --> STATUS &
  LMU_Eval: LMU_DATA --> STATUS &
  POS_Eval: POS_DATA --> STATUS &
  N_DB: NAT &
  N_UE: NAT &
  N_LMU: NAT &
  N_POS: NAT &
  DB_Eval(DB_FAIL) = UNRECOV &
  UE_Eval(UE_FAIL) = UNRECOV &
  LMU_Eval(LMU_FAIL) = UNRECOV &
  POS_Eval(POS_FAIL) = UNRECOV &
  LMU_Eval(LMU_OK) = OK

END
```

```
REFINEMENT
  SDirector

REFINES
  Main

SEES
  Comp_data

SETS
 SERVICE = {SD,DB,UE,LMU,POS,CALC}


VARIABLES
  inp_chan, input, out_chan, output, curr_service, handling_flag,
  dbdata, uedata, lmudata, posdata, n_db, n_ue, n_lmu, n_pos

INVARIANT
  curr_service : SERVICE &
  handling_flag : BOOL &
  dbdata : DB_DATA &
  uedata : UE_DATA &
  lmudata : LMU_DATA &
  posdata : POS_DATA &
  (curr_service : SERVICE-{SD} => not(input = INPUT_NIL)) &
  (curr_service=CALC => posdata:POS_DATA-{POS_NIL}) &
  (curr_service=CALC => handling_flag=FALSE) &
  (handling_flag=TRUE => not(input=INPUT_NIL)) &
  n_db:NAT & n_db <= N_DB &
  n_ue:NAT & n_ue <= N_UE &
  n_lmu:NAT & n_lmu <= N_LMU &
  n_pos:NAT & n_pos <= N_POS

INITIALISATION
  inp_chan, input := INPUT_NIL, INPUT_NIL ||
  out_chan, output := POS_NIL, POS_NIL ||
  curr_service, handling_flag := SD,FALSE ||
  dbdata,uedata,lmudata,posdata := DB_NIL,UE_NIL,LMU_NIL,POS_NIL ||
  n_db,n_ue,n_lmu,n_pos := N_DB,N_UE,N_LMU,N_POS

OPERATIONS

env_write =
  SELECT inp_chan = INPUT_NIL
  THEN
    inp_chan :: INPUT_DATA - {INPUT_NIL}
  END;

read =
  SELECT not(inp_chan = INPUT_NIL) & (input = INPUT_NIL)
  THEN
    input,inp_chan := inp_chan,INPUT_NIL ||
    handling_flag := TRUE ||
```

```
    curr_service := SD
  END;

db =
  SELECT curr_service = DB
  THEN
    handling_flag := TRUE
  END;

ue =
  SELECT curr_service = UE
  THEN
    handling_flag := TRUE
  END;

lmu =
  SELECT curr_service = LMU
  THEN
    handling_flag := TRUE
  END;

pos =
  SELECT curr_service = POS
  THEN
    handling_flag := TRUE
  END;


/* can be splitted into several handlers */
handle =
  SELECT handling_flag = TRUE
  THEN
    IF curr_service = SD
    THEN
      curr_service := DB
    ELSIF curr_service = DB
    THEN
      dbdata :: DB_DATA-{DB_NIL};
      IF DB_Eval(dbdata) = OK
      THEN
        curr_service := UE
      ELSIF DB_Eval(dbdata) = RECOV & (n_db > 0)
      THEN
        n_db := n_db-1
      ELSE
        posdata,curr_service := POS_FAIL,CALC
      END
    ELSIF curr_service = UE
    THEN
      uedata :: UE_DATA-{UE_NIL};
      IF UE_Eval(uedata) = OK
      THEN
        curr_service := LMU
      ELSIF UE_Eval(uedata) = RECOV & (n_ue > 0)
      THEN
```

```
      n_ue := n_ue-1
    ELSE
      posdata,curr_service := POS_FAIL,CALC
    END
  ELSIF curr_service = LMU
  THEN
    lmudata :: LMU_DATA-{LMU_NIL};
    IF LMU_Eval(lmudata) = OK
    THEN
      curr_service := POS
    ELSIF LMU_Eval(lmudata) = RECOV & (n_lmu > 0)
    THEN
      n_lmu := n_lmu-1
    ELSE
      posdata,curr_service := POS_FAIL,CALC
    END
  ELSIF curr_service = POS
  THEN
    posdata :: POS_DATA-{POS_NIL};
    IF POS_Eval(posdata) = OK
    THEN
      curr_service := CALC
    ELSIF POS_Eval(posdata) = RECOV & (n_pos > 0)
    THEN
      n_pos := n_pos-1
    ELSE
      posdata,curr_service := POS_FAIL,CALC
    END
  END ||
  handling_flag := FALSE
 END;


calculate =
 SELECT not(input = INPUT_NIL) & (output = POS_NIL) & (curr_service=CALC)
 THEN
  output,input := posdata,INPUT_NIL ||
  curr_service := SD
 END;

write =
 SELECT not(output = POS_NIL) & (out_chan = POS_NIL)
 THEN
  out_chan,output := output,POS_NIL
 END;

env_read =
 SELECT not(out_chan = POS_NIL)
 THEN
  out_chan := POS_NIL
 END

END
```

```
REFINEMENT
  SDirectorRef

REFINES
  SDirector

SEES
  Comp_data

INCLUDES
  DB_Comp, UE_Comp, SAS_Comp

VARIABLES
  inp_chan, input, out_chan, output, curr_service, handling_flag,
  dbdata, uedata, posdata, n_db, n_ue, n_lmu, n_pos

INVARIANT
  curr_service : SERVICE &
  handling_flag : BOOL &
  dbdata : DB_DATA &
  uedata : UE_DATA &
  posdata : POS_DATA &
  (curr_service = UE => not(dbdata=DB_NIL)) &
  (curr_service = LMU => not(uedata=UE_NIL)) &
  (curr_service = POS => not(uedata=UE_NIL)) &
  n_db:NAT & n_db <= N_DB &
  n_ue:NAT & n_ue <= N_UE &
  n_lmu:NAT & n_lmu <= N_LMU &
  n_pos:NAT & n_pos <= N_POS

INITIALISATION
  inp_chan, input := INPUT_NIL, INPUT_NIL ||
  out_chan, output := POS_NIL, POS_NIL ||
  curr_service, handling_flag := SD,FALSE ||
  dbdata,uedata,posdata := DB_NIL,UE_NIL,POS_NIL ||
  n_db,n_ue,n_lmu,n_pos := N_DB,N_UE,N_LMU,N_POS

OPERATIONS

env_write =
  SELECT inp_chan = INPUT_NIL
  THEN
    inp_chan :: INPUT_DATA - {INPUT_NIL}
  END;

read =
  SELECT not(inp_chan = INPUT_NIL) & (input = INPUT_NIL)
  THEN
    input,inp_chan := inp_chan,INPUT_NIL ||
    handling_flag := TRUE ||
    curr_service := SD
  END;
```

```
db =
 SELECT curr_service = DB & (db_inp_chan=INPUT_NIL)
 THEN
  db_write_ichan(input);
  handling_flag := TRUE
 END;

ue =
 SELECT curr_service = UE & (ue_inp_chan=DB_NIL)
 THEN
  ue_write_ichan(dbdata);
  handling_flag := TRUE
 END;

lmu =
 SELECT curr_service = LMU
 THEN
  handling_flag := TRUE
 END;

pos =
 SELECT curr_service = POS & (sas_inp_chan=UE_NIL)
 THEN
  sas_write_ichan(uedata);
  handling_flag := TRUE
 END;

handle =
 SELECT handling_flag = TRUE &
     (((curr_service = DB) & not(db_out_chan = DB_NIL)) or
      ((curr_service = UE) & not(ue_out_chan = UE_NIL)) or
      ((curr_service = POS) & not(sas_out_chan = POS_NIL)) or
      (curr_service = LMU))
 THEN
  IF curr_service = SD
  THEN
    curr_service := DB
  ELSIF curr_service = DB
  THEN
    dbdata <-- db_read_ochan;
    IF DB_Eval(dbdata) = OK
    THEN
     curr_service := UE
    ELSIF DB_Eval(dbdata) = RECOV & (n_db > 0)
    THEN
     n_db := n_db-1
    ELSE
     posdata,curr_service := POS_FAIL,CALC
    END
  ELSIF curr_service = UE
  THEN
    uedata <-- ue_read_ochan;
    IF UE_Eval(uedata) = OK
    THEN
     curr_service := LMU
```

```
      ELSIF UE_Eval(uedata) = RECOV & (n_ue > 0)
      THEN
        n_ue := n_ue-1
      ELSE
        posdata,curr_service := POS_FAIL,CALC
      END
    ELSIF curr_service = LMU
    THEN
        curr_service := POS
    ELSIF curr_service = POS
    THEN
      posdata <-- sas_read_ochan;
      IF POS_Eval(posdata)=UNRECOV
      THEN
        posdata := POS_FAIL
      END;
      curr_service := CALC
    END ||
    handling_flag := FALSE
  END;


calculate =
  SELECT not(input = INPUT_NIL) & (output = POS_NIL) & (curr_service=CALC)
  THEN
    output,input := posdata,INPUT_NIL ||
    curr_service := SD
  END;

write =
  SELECT not(output = POS_NIL) & (out_chan = POS_NIL)
  THEN
    out_chan,output := output,POS_NIL
  END;

env_read =
  SELECT not(out_chan = POS_NIL)
  THEN
    out_chan := POS_NIL
  END

END
```

```
MACHINE
  DB_Comp

SEES
  Comp_data
VARIABLES
  db_inp_chan, db_input, db_out_chan, db_output

INVARIANT
  db_inp_chan : INPUT_DATA &
  db_input : INPUT_DATA &
  db_out_chan : DB_DATA &
  db_output : DB_DATA
INITIALISATION
  db_inp_chan, db_input := INPUT_NIL, INPUT_NIL ||
  db_out_chan, db_output := DB_NIL, DB_NIL

OPERATIONS

db_write_ichan(inp) =
  PRE inp:INPUT_DATA & not(inp=INPUT_NIL) & (db_inp_chan=INPUT_NIL)
  THEN
    db_inp_chan := inp
  END;

db_read =
  SELECT not(db_inp_chan = INPUT_NIL) & (db_input = INPUT_NIL)
  THEN
    db_input,db_inp_chan := db_inp_chan,INPUT_NIL
  END;

db_calculate =
  SELECT not(db_input = INPUT_NIL) & (db_output = DB_NIL)
  THEN
    CHOICE
      db_output :: DB_DATA - {DB_NIL,DB_FAIL}
    OR
      db_output := DB_FAIL
    END ||
    db_input := INPUT_NIL
  END;

db_write =
  SELECT not(db_output = DB_NIL) & (db_out_chan = DB_NIL)
  THEN
    db_out_chan,db_output := db_output,DB_NIL
  END;

db_out <-- db_read_ochan =
  PRE not(db_out_chan = DB_NIL)
  THEN
    db_out,db_out_chan := db_out_chan,DB_NIL
  END
END
```

```
MACHINE
  UE_Comp

SEES
  Comp_data

VARIABLES
  ue_inp_chan, ue_input, ue_out_chan, ue_output

INVARIANT
  ue_inp_chan : DB_DATA &
  ue_input : DB_DATA &
  ue_out_chan : UE_DATA &
  ue_output : UE_DATA

INITIALISATION
  ue_inp_chan,ue_input := DB_NIL,DB_NIL ||
  ue_out_chan, ue_output := UE_NIL, UE_NIL
OPERATIONS

ue_write_ichan(inp) =
  PRE inp:DB_DATA & not(inp=DB_NIL) & (ue_inp_chan=DB_NIL)
  THEN
    ue_inp_chan := inp
  END;

ue_read =
  SELECT not(ue_inp_chan = DB_NIL) & (ue_input = DB_NIL)
  THEN
    ue_input,ue_inp_chan := ue_inp_chan,DB_NIL
  END;

ue_calculate =
  SELECT not(ue_input = DB_NIL) & (ue_output = UE_NIL)
  THEN
    CHOICE
      ue_output :: UE_DATA - {UE_NIL,UE_FAIL}
    OR
      ue_output := UE_FAIL
    END ||
    ue_input := DB_NIL
  END;

ue_write =
  SELECT not(ue_output = UE_NIL) & (ue_out_chan = UE_NIL)
  THEN
    ue_out_chan,ue_output := ue_output,UE_NIL
  END;

ue_out <-- ue_read_ochan =
  PRE not(ue_out_chan = UE_NIL)
  THEN
    ue_out,ue_out_chan := ue_out_chan,UE_NIL
  END
END
```

```
MACHINE
  SAS2_Comp

SEES
  Comp_data

SETS
  SAS_SERVICE = {SAS,SAS_LMU,SAS_POS,SAS_CALC}

VARIABLES
  sas_inp_chan, sas_input, sas_out_chan, sas_output,
  sas_curr_service, sas_handling_flag, sas_lmudata, sas_posdata,
  sas_n_lmu, sas_n_pos

INVARIANT
  sas_inp_chan : UE_DATA &
  sas_input : UE_DATA &
  sas_out_chan : POS_DATA &
  sas_output : POS_DATA &
  sas_curr_service : SAS_SERVICE &
  sas_handling_flag : BOOL &
  sas_lmudata : LMU_DATA &
  sas_posdata : POS_DATA &
  sas_n_lmu : NAT & sas_n_lmu <= N_LMU &
  sas_n_pos : NAT & sas_n_pos <= N_POS &
  (sas_curr_service=SAS_CALC => POS_Eval(sas_posdata):{OK,UNRECOV}) &
  (not(sas_output = POS_NIL) => POS_Eval(sas_output):{OK,UNRECOV}) &
  (not(sas_out_chan = POS_NIL) => POS_Eval(sas_out_chan):{OK,UNRECOV})


INITIALISATION
  sas_inp_chan, sas_input := UE_NIL, UE_NIL ||
  sas_out_chan, sas_output := POS_NIL, POS_NIL ||
  sas_curr_service, sas_handling_flag := SAS, FALSE ||
  sas_lmudata, sas_posdata := LMU_NIL, POS_NIL ||
  sas_n_lmu, sas_n_pos := N_LMU, N_POS

OPERATIONS

sas_write_ichan(inp) =
  PRE inp:UE_DATA & not(inp=UE_NIL) & (sas_inp_chan=UE_NIL)
  THEN
    sas_inp_chan := inp
  END;

sas_read =
  SELECT not(sas_inp_chan = UE_NIL) & (sas_input = UE_NIL)
  THEN
    sas_input,sas_inp_chan := sas_inp_chan,UE_NIL ||
    sas_curr_service := SAS
  END;

sas_lmu =
  SELECT sas_curr_service = SAS_LMU
  THEN
```

```
    sas_handling_flag := TRUE
  END;

sas_pos =
  SELECT sas_curr_service = SAS_POS
  THEN
    sas_handling_flag := TRUE
  END;

sas_handle =
  SELECT sas_handling_flag = TRUE
  THEN
    IF sas_curr_service = SAS
    THEN
      sas_curr_service := SAS_LMU
    ELSIF sas_curr_service = SAS_LMU
    THEN
      ANY lmudata WHERE lmudata : LMU_DATA
      THEN
        sas_lmudata := lmudata ||
        IF LMU_Eval(lmudata) = OK
        THEN
          sas_curr_service := SAS_POS
        ELSIF LMU_Eval(lmudata) = RECOV & (sas_n_lmu > 0)
        THEN
          sas_n_lmu := sas_n_lmu-1
        ELSE
          sas_posdata,sas_curr_service := POS_FAIL,SAS_CALC
        END
      END
    ELSIF sas_curr_service = SAS_POS
    THEN
      ANY posdata WHERE posdata : POS_DATA
      THEN
        IF ((POS_Eval(posdata) = RECOV) & (sas_n_pos = 0)) or
           POS_Eval(posdata) = UNRECOV
        THEN
          sas_posdata := POS_FAIL
        ELSE
          sas_posdata := posdata
        END
        ||
        IF POS_Eval(posdata) = OK
        THEN
          sas_curr_service := SAS_CALC
        ELSIF POS_Eval(posdata) = RECOV & (sas_n_pos > 0)
        THEN
          sas_n_pos := sas_n_pos-1
        ELSE
          sas_curr_service := SAS_CALC
        END
      END
    END ||
    sas_handling_flag := FALSE
  END;
```

```
sas_calculate =
  SELECT not(sas_input = UE_NIL) & (sas_output = POS_NIL) &
         (sas_curr_service=SAS_CALC)
  THEN
    sas_output,sas_input := sas_posdata,UE_NIL ||
    sas_curr_service := SAS
  END;

sas_write =
  SELECT not(sas_output = POS_NIL) & (sas_out_chan = POS_NIL)
  THEN
    sas_out_chan,sas_output := sas_output,POS_NIL
  END;

sas_out <-- sas_read_ochan =
  PRE not(sas_out_chan = POS_NIL)
  THEN
    sas_out,sas_out_chan := sas_out_chan,POS_NIL
  END

END
```

```
REFINEMENT
  SAS_CompRef

REFINES
  SAS_Comp

SEES
  Comp_data

INCLUDES
  LMU_Comp, POS_Comp

VARIABLES
  sas_inp_chan, sas_input, sas_out_chan, sas_output,
  sas_curr_service, sas_handling_flag, sas_lmudata, sas_posdata,
  sas_n_lmu, sas_n_pos


INVARIANT
  sas_inp_chan : UE_DATA &
  sas_input : UE_DATA &
  sas_out_chan : POS_DATA &
  sas_output : POS_DATA &
  sas_curr_service : SAS_SERVICE &
  sas_handling_flag : BOOL &
  sas_lmudata : LMU_DATA &
  sas_posdata : POS_DATA &
  sas_n_lmu : NAT &
  sas_n_pos : NAT &
  (sas_curr_service = SAS_LMU => not(sas_input=UE_NIL)) &
  (sas_curr_service = SAS_POS => not(sas_lmudata=LMU_NIL))


INITIALISATION
  sas_inp_chan, sas_input := UE_NIL, UE_NIL ||
  sas_out_chan, sas_output := POS_NIL, POS_NIL ||
  sas_curr_service, sas_handling_flag := SAS, FALSE ||
  sas_lmudata, sas_posdata := LMU_NIL, POS_NIL ||
  sas_n_lmu, sas_n_pos := N_LMU,N_POS

OPERATIONS

sas_write_ichan(inp) =
  PRE inp:UE_DATA & not(inp=UE_NIL) & (sas_inp_chan=UE_NIL)
  THEN
    sas_inp_chan := inp
  END;

sas_read =
  SELECT not(sas_inp_chan = UE_NIL) & (sas_input = UE_NIL)
  THEN
    sas_input,sas_inp_chan := sas_inp_chan,UE_NIL ||
    sas_curr_service := SAS
```

```
 END;

sas_lmu =
 SELECT sas_curr_service = SAS_LMU & (lmu_inp_chan=UE_NIL)
 THEN
   lmu_write_ichan(sas_input);
   sas_handling_flag := TRUE
 END;

sas_pos =
 SELECT sas_curr_service = SAS_POS & (pos_inp_chan=LMU_NIL)
 THEN
   pos_write_ichan(sas_lmudata);
   sas_handling_flag := TRUE
 END;

sas_handle =
 SELECT sas_handling_flag = TRUE &
       (((sas_curr_service = SAS_LMU) & not(lmu_out_chan=LMU_NIL)) or
        ((sas_curr_service = SAS_POS) & not(pos_out_chan=POS_NIL)))
 THEN
   IF sas_curr_service = SAS
   THEN
     sas_curr_service := SAS_LMU
   ELSIF sas_curr_service = SAS_LMU
   THEN
     sas_lmudata <-- lmu_read_ochan;
     IF LMU_Eval(sas_lmudata) = OK
     THEN
       sas_curr_service := SAS_POS
     ELSIF LMU_Eval(sas_lmudata) = RECOV & (sas_n_lmu > 0)
     THEN
       sas_n_lmu := sas_n_lmu-1
     ELSE
       sas_posdata,sas_curr_service := POS_FAIL,SAS_CALC
     END
   ELSIF sas_curr_service = SAS_POS
   THEN
     sas_posdata <-- pos_read_ochan;
     IF POS_Eval(sas_posdata) = OK
     THEN
       sas_curr_service := SAS_CALC
     ELSIF POS_Eval(sas_posdata) = RECOV & (sas_n_pos > 0)
     THEN
       sas_n_pos := sas_n_pos-1
     ELSE
       sas_posdata,sas_curr_service := POS_FAIL,SAS_CALC
      END
   END ||
   sas_handling_flag := FALSE
 END;

sas_calculate =
 SELECT not(sas_input = UE_NIL) & (sas_output = POS_NIL) &
       (sas_curr_service=SAS_CALC)
```

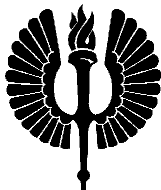
```
  THEN
    sas_output,sas_input := sas_posdata,UE_NIL ||
    sas_curr_service := SAS
  END;

sas_write =
  SELECT not(sas_output = POS_NIL) & (sas_out_chan = POS_NIL)
  THEN
    sas_out_chan,sas_output := sas_output,POS_NIL
  END;

sas_out <-- sas_read_ochan =
  PRE not(sas_out_chan = POS_NIL)
  THEN
    sas_out,sas_out_chan := sas_out_chan,POS_NIL
  END

END
```

```
MACHINE
  LMU_Comp

SEES
  Comp_data

VARIABLES
  lmu_inp_chan, lmu_input, lmu_out_chan, lmu_output

INVARIANT
  lmu_inp_chan : UE_DATA &
  lmu_input : UE_DATA &
  lmu_out_chan : LMU_DATA &
  lmu_output : LMU_DATA

INITIALISATION
  lmu_inp_chan, lmu_input := UE_NIL, UE_NIL ||
  lmu_out_chan, lmu_output := LMU_NIL, LMU_NIL
OPERATIONS

lmu_write_ichan(inp) =
  PRE inp:UE_DATA & not(inp=UE_NIL) & (lmu_inp_chan=UE_NIL)
  THEN
    lmu_inp_chan := inp
  END;
lmu_read =
  SELECT not(lmu_inp_chan = UE_NIL) & (lmu_input = UE_NIL)
  THEN
    lmu_input,lmu_inp_chan := lmu_inp_chan,UE_NIL
  END;
lmu_calculate =
  SELECT not(lmu_input = UE_NIL) & (lmu_output = LMU_NIL)
  THEN
    CHOICE
      lmu_output :: LMU_DATA - {LMU_NIL,LMU_FAIL}
    OR
      lmu_output := LMU_FAIL
    END ||
    lmu_input := UE_NIL
  END;

lmu_write =
  SELECT not(lmu_output = LMU_NIL) & (lmu_out_chan = LMU_NIL)
  THEN
    lmu_out_chan,lmu_output := lmu_output,LMU_NIL
  END;

lmu_out <-- lmu_read_ochan =
  PRE not(lmu_out_chan = LMU_NIL)
  THEN
    lmu_out,lmu_out_chan := lmu_out_chan,LMU_NIL
  END
END
```

```
MACHINE
  POS_Comp

SEES
  Comp_data

VARIABLES
  pos_inp_chan, pos_input, pos_out_chan, pos_output

INVARIANT
  pos_inp_chan : LMU_DATA &
  pos_input : LMU_DATA &
  pos_out_chan : POS_DATA &
  pos_output : POS_DATA

INITIALISATION
  pos_inp_chan, pos_input := LMU_NIL, LMU_NIL ||
  pos_out_chan, pos_output := POS_NIL, POS_NIL

OPERATIONS

pos_write_ichan(inp) =
  PRE inp:LMU_DATA & not(inp=LMU_NIL) & (pos_inp_chan=LMU_NIL)
  THEN
    pos_inp_chan := inp
  END;
pos_read =
  SELECT not(pos_inp_chan = LMU_NIL) & (pos_input = LMU_NIL)
  THEN
    pos_input,pos_inp_chan := pos_inp_chan,LMU_NIL
  END;
pos_calculate =
  SELECT not(pos_input = LMU_NIL) & (pos_output = POS_NIL)
  THEN
    CHOICE
      pos_output :: POS_DATA - {POS_NIL,POS_FAIL}
    OR
      pos_output := POS_FAIL
    END ||
    pos_input := LMU_NIL
  END;
pos_write =
  SELECT not(pos_output = POS_NIL) & (pos_out_chan = POS_NIL)
  THEN
    pos_out_chan,pos_output := pos_output,POS_NIL
  END;

pos_out <-- pos_read_ochan =
  PRE not(pos_out_chan = POS_NIL)
  THEN
    pos_out,pos_out_chan := pos_out_chan,POS_NIL
  END
END
```

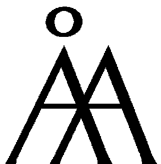
Turku Centre *for* Computer Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 952-12-1564-X
ISSN 1239-1891