# Formal Model-Driven Development of Communicating Systems


Linas Laibinis[1], Elena Troubitsyna[1], Sari Leppänen[2], Johan Lilius[1], Qaisar Malik[1]

[1]Åbo Akademi, Department of Computer Science,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
(Linas.Laibinis, Elena.Troubitsyna,
Johan.Lilius, Qaisar.Malik)@abo.fi
[2]Nokia Research Center, Computing Architectures Laboratory,
P.O. Box 407, 00045, Helsinki, Finland
Sari.Leppanen@nokia.com



**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. 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.


## 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 in turn 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) [10] 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 Center has developed the design method *Lyra* [8] – 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.

From the beginning Lyra has been developed in such a way that it would be possible to bring formal methods (such as program refinement, model checking, model-based testing etc.) into more extensive industrial use. A formalisation of the Lyra development would allow us to ensure correctness of system design via automatic and formally verified construction. The achievement of such a formalisation would be considered as significant added value for industry.

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 [1,13] – a formal framework with automatic tool support. While developing a system by refinement, we start from an abstract specification and gradually incorporate implementation details into it until 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 a 3GPP positioning system [15,16].

## 2 Lyra: Service-Based Development of Communicating Systems

**Overview of Lyra.** Lyra [8] is a model-driven and component-based design method for the development of communicating systems and communication protocols. It has been developed in the 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 prestandardisation to final implementation. It has been successfully applied in large-scale UML2-based industrial software development, e.g., for specification of architecture for several network components, standardisation of 3GPP protocols, implementation of several network protocols etc.

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 the *Service Distribution* phase, the logical architecture of services is distributed over a given platform architecture. Finally, in the *Service Implementation* phase, the structural elements are adjusted and integrated into 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.

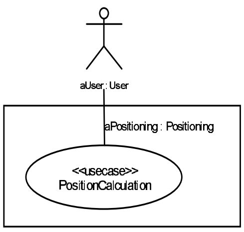
**Lyra by example.** We model part of a Third Generation Partnership Project (3GPP) positioning system [15,16]. The positioning system provides positioning services to calculate the physical location of a given item of user equipment (UE) in a mobile network. We focus on Position Calculation Application Part (PCAP) – a part of the positioning system allowing communication in a 3GPP network. 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 [15,16].

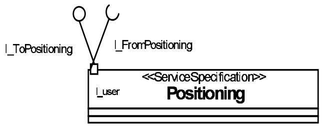
The Service Specification phase starts from creating a domain model of the system. It describes the system with the included system-level services and different types of external users. Each association connecting an external user and a system level service corresponds to a logical interface. For the 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 presentation of these 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.

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 *(Refer-*

*enceLMU)* 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 the Algorithm service provider (*Algorithm),* which 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 the Lyra development – *Service Decomposition* – we focus on specifying service execution according to the identified stages.



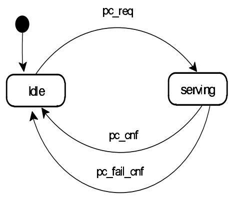
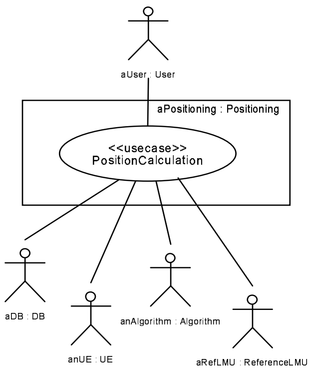




**Fig.1a.** Domain model **Fig.1b** PSAP of Positioning **Fig.1c** State diagram

In the Service Decomposition phase, we introduce the 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 the corresponding 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 a UE position can be decomposed as shown in Fig.2c. The sequence diagrams (omitted here) are created to model 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 the functionalities related to a single execution stage or other logical piece of functionality.



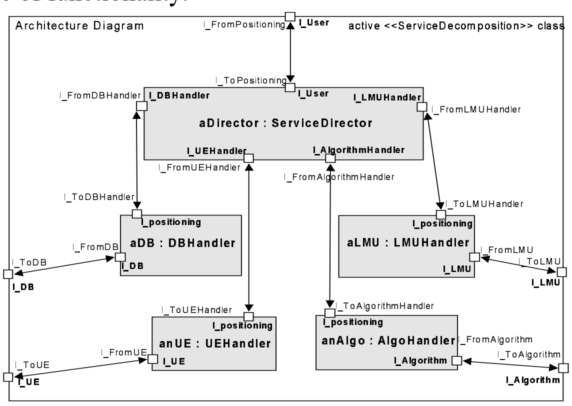




**Fig.2a.**Domain model **Fig. 2d.** PositionCalculation functional architecture

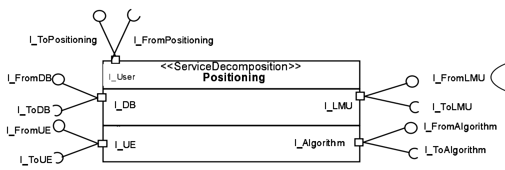



Fig.**2b.** PSAP and USAPs of Positioning

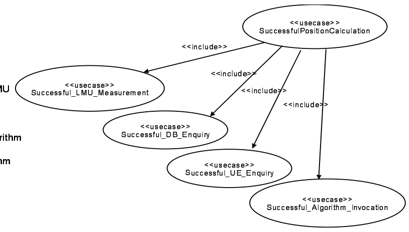



Fig.2c. Use case decomposition

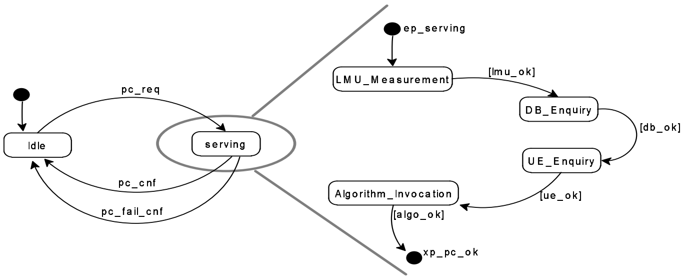



Fig.**2e.** ServiceDirector: PSAP communication and execution control

In Fig.2d we present the architecture diagram of the *Positioning* system. Here *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 state submachine *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. In their turns, these machines (omitted here) include specifications of the specific PSAP-USAP communications.

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. The signalling network protocols are used for communication between the service components in distant network elements.

In Fig.3a we illustrate the physical structure of the distributed positioning system. Here *Positioning_RND* and *Positioning_SAS* represent network elements in a UMTS network. The Protocol Data Unit (PDU) interface *Iupc* is used in communication between the network elements. We map the functional architecture to the given 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 the 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 [8, 15, 16].

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.

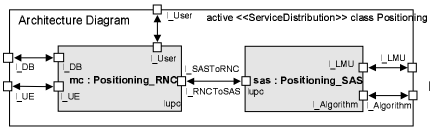



Fig.3a. Architecture of service

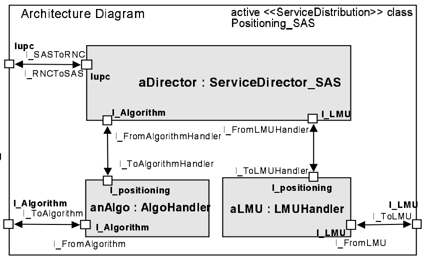



Fig.3b. Architecture of Positioning_SAS

## 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 [4,9]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [13], 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 system, or a part of a system.

The B method provides us with mechanisms for structuring the system architecture by modularisation. A module is represented as an *abstract machine*. An abstract machine encapsulates state (a set of program variables) and operations of the specification. The abstract machines 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.

Each abstract 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. The operations in B can be 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 execution of the corresponding operation is suspended, i.e., the operation is in waiting mode until cond becomes true. Such B operations are suitable for specifying system reactions on events, i.e., for modelling common reactive systems.

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, the conditional statement and sequential composition have the standard meaning. The remaining constructs allow us to model nondeterministic or parallel behaviour in a specification. For example, x :: T denotes a nondeterministic assignment where any value from set T can be assigned to variable x. Usually such statements 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].

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

**Modelling a Service Component in B.** Above we have described a service component as a coherent piece of functionality that provides its services to a service consumer via PSAP(s). 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(s) and sends the results of service execution.

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 a service. Our specification of *ACAM* is merely a statement nondeterministically generating results of the 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 systemic 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 the 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 the execution of read, 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 constants INPUT_NIL and OUT_NIL to model the absence of data, i.e., the empty channel . The operations discussed above model the communicational (*ACM)* part of *ACC.*

```
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
```

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 the corresponding protocols for transferring them.

In the next section we demonstrate how Lyra development flow can be formalized as refinement and decomposition of an abstract communicating component (ACC).

## 4 Formal Service-Oriented Development

As described in Section 2, usually a service component is represented as an active class with the PSAP(s) attached to it via the port(s). The state diagram depicts the 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 *ACC* pattern proposed in Section 3. The general principle of translation is shown in Fig.4.

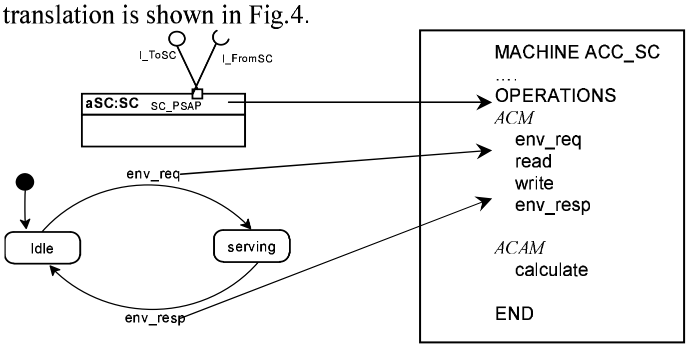



**Fig.4.** Translating UML2 model into the ACC pattern

The UML2 description of PSAP of the service component *SC* is translated into the communicational (*ACM)* part of the machine ACC_SC specifying *SC* according to the *ACC* pattern. The functional (*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:

1. the service execution is decomposed into a number of stages (or subservices).
2. 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 the execution by enquiring the required subservices from the external entities according to the 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 such decomposition can be represented as a refinement of our abstract pattern *ACC* instantiated to model *SC*.

This decomposition step focuses on refinement of the functional (*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 this paper, due to the lack of space, we omit the presentation of failures of service provision and error recovery while specifying *Service Director*. The detailed description of this can be found in the accompanying technical report [5].

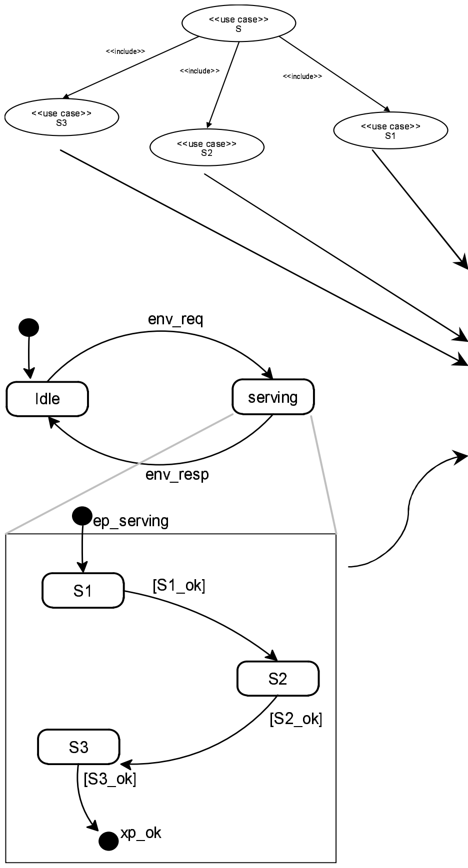
To derive the pattern for translating UML2 diagrams modelling the functional architecture and the platform-distributed service architecture at these two phases, we should consider two general cases:

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 the 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 initial service component SC looks as shown in Fig.6. The service distribution architecture diagram for the first case is given in Fig.7.

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 the 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 communicational (*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 the external service components, in this refinement step we also define the mechanisms for communicating with them. We refine the operation director to specify the communication on USAPs. Namely, we replace the nondeterministic assignments modelling stages of the 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




```
REFINEMENT ACC_R1_SC
REFINES ACC_SC
VARIABLES
  curr_service, handling_flag
INVARIANT
  curr_service : {SD, S1, S2,S3, CALC} &
  handling_flag : BOOL & ...

INITIALISATION
  curr_service, handling_flag := SD,FALSE || ...
OPERATIONS
    ACM...
```

```
ACAM'
S1 = SELECT curr_service = S1
        THEN  handling_flag := TRUE
        END;
S2 = ...
S3 = ...
director =
 SELECT handling_flag = TRUE THEN
   IF curr_service = SD THEN
     curr_service := S1
   ELSIF curr_service = S1 THEN
     S1_data :: S1_DATA-{S1_NIL};
      curr_service := S2
   ELSIF curr_service = S2 ...
         ELSIF curr_service = S3
         THEN ...
             curr_service := CALC
   END ||
   handling_flag := FALSE
 END;
calculate =
 SELECT (curr_service=CALC) & ...
 THEN
    output,input := OUT_data,INPUT_NIL ||
    curr_service := SD
 END;

END
```

**Fig.5.** Service decomposition and refinement

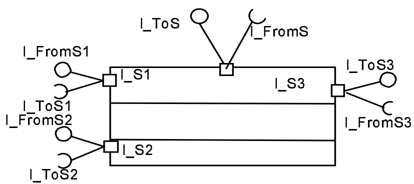



**Fig.6.** Service component with USAPs

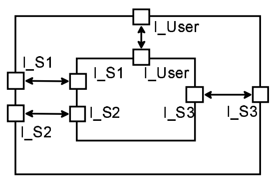



**Fig.7.** Architecture diagram (case 1)

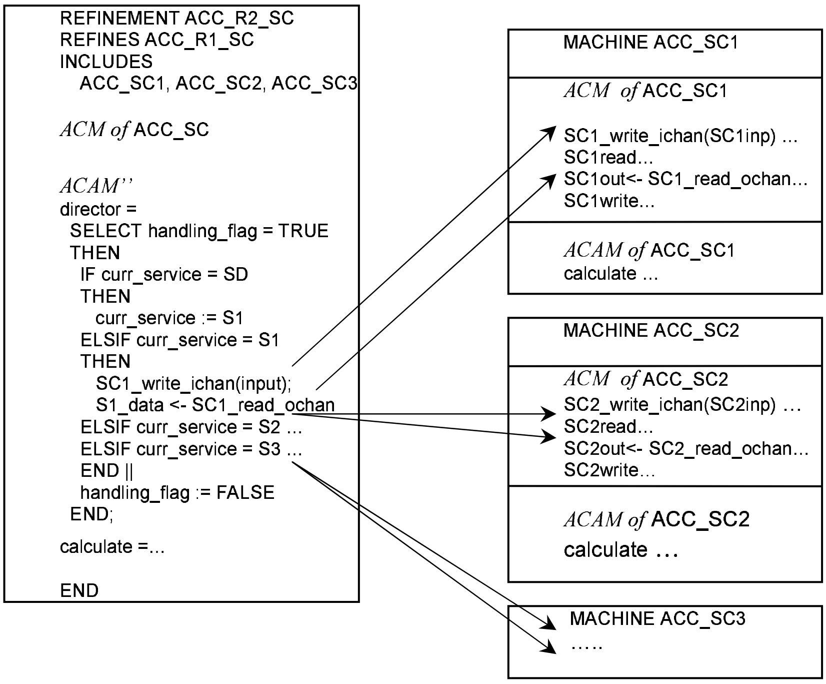
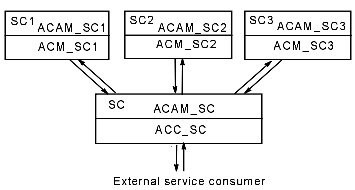



**Fig.8.** Refinement at *Service Decomposition* and *Service Distribution* phases

results from the output channel of this machine, e.g., SC1_read_ochan. Graphically this arrangement is depicted in Fig.9.

Modelling case (2) 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 the *SC2* and *SC3* services. The architecture diagram depicting the overall arrangement is shown in Fig.10.



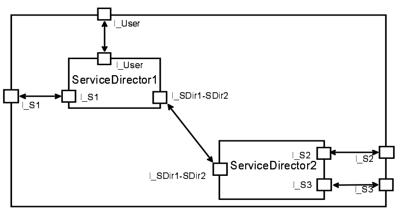




**Fig.9.** Architecture of formal specification **Fig.10.** Architecture diagram (case 2)

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 *S1*. After the result of *S1* 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 the "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 the functional (*ACAM)* part 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.

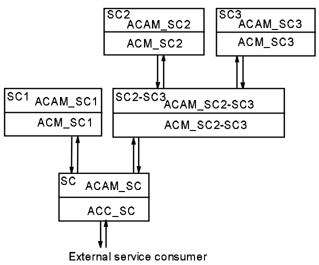



**Fig.11.** Architecture (case 2)

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

**Discussion.** In the proposed approach we have used our B formalisation of Lyra to verify correctness of the Lyra decomposition and distribution phases. We have done this by introducing generic patterns for communicating service components and then associating the Lyra development steps with the corresponding B refinements on these patterns. In development of real systems we merely have to establish by proof that the corresponding components in a specific functional or network architecture are valid instantiations of these patterns. All together this constitutes a basis for automating industrial design flow of communicating systems.

The decomposition model that we have used for testing our approach is still relatively simple. As a result, all refinement steps were automatically proved by AtelierB –a tool supporting B. While describing the formalisation of Lyra in B, we considered only the sequential model of service execution. However, parallel execution of services is also a valid interpretation of the considered UML2 models. Currently we are working on extending our B models to include parallel execution of services. Furthermore, we will incorporate more sophisticated fault tolerance mechanisms (e.g., different types of fault recovery procedures) into our models. We foresee that such extensions will make automatic proof of model refinements more difficult. However, by developing generic proof strategies, we will try to achieve high degree of automation in formal verification of our models.

## 5. Conclusions

In this paper we proposed a formal approach to development of communicating distributed systems. Our approach formalizes Lyra [8] – the UML2-based design methodology adopted in Nokia. The formalization is done within the B Method [1,13] – a 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. Such automation would enable 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 [8]. However, since 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 to the consideration of the recent research conducted within the B Method.

Treharne et al. [14] 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 active research going on translating UML to B [3,6,7,11,12]. Among these, the most notable is research conducted by Snook and Butler [11] 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 the U2B developers 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. We are already working on creating the Lyra UML2 metamodel, which will assist us in achieving this goal. 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). We are also grateful to anonymous reviewers for their very helpful comments.

## 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 H. Treharne et al (Eds.), Formal Specification and Development in Z and B:,UK, 2005.
3. P.Facon, et al. *Combining UML with the B formal method for the specification of database applications.* Research report, CEDRIC laboratory, Paris, 1999.
4. P.Behm, et al. *METEOR: A successful application of B in a large project*. In Wing et all (editors), Proc. of the World Congress on Formal Methods. LNCS 1709, Springer, 1999.
5. L.Laibinis, E.Troubitsyna, S.Leppänen, J.Lilius, and Q.Malik. *Formal Model-Driven Development of Communicating Systems*. TUCS Technical Report No. 691. Finland, 2005.
6. K.Lano, D.Clark, and K.Adroutsopoulos. *UML to B: Formal Verification of Object-Oriented Models*. In E.A.Boiten et al (Eds.): Integrated Formal Methods,. Springer, LNCS 2999.
7. H.LeDang and J.Souquieres. *Integrating UML and B specification techniques*. In Proc. of the Workshop on Integrating Diagrammatic and Formal Specification Techniques, 2001.
8. S.Leppänen, M.Turunen, and I.Oliver. *Application Driven Methodology for Development of Communicating Systems*.Forum on Specification and Design Languages. France, 2004.
9. MATISSE Handbook for Correct Systems Construction. 2003. http://www.esil.univ-mrs.fr/~spc/matisse/Handbook/.
10. J.Rumbaugh, I.Jacobson, and G.Booch. *The Unified Modelling Language* Reference Manual. Addison-Wesley, 1998.
11. 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. Springer, 2004.
12. C.Snook and M.Waldén. *Use of U2B for Specifying B Action Systems.* In Proc. of Workshop on Refinement of Critical Systems: Methods, Tools and Experience, France, 2002.
13. Steria, Aix-en-Provence, France. Atelier B, User and Reference Manuals, 2001. Available at http://www.atelierb.societe.com/index uk.html
14. H.Treharne et al. *Composing Specifications Using Communication*, in D. Bert et al (Eds.), Proc. of Formal Specification and Development in Z and B, Finland. Springer, 2003.
15. 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
16. 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