Linas Laibinis | Elena Troubitsyna | Sari Leppänen | Johan Lilius | Qaisar Ahmad Malik

Formal Service-Oriented Development of Fault Tolerant Communicating Systems

Turku Centre *for* Computer Science

TUCS Technical Report
No 764, April 2006

Formal Service-Oriented Development of Fault Tolerant Communicating Systems

Linas Laibinis
Elena Troubitsyna
Åbo Akademi University, Department of Information Technologies,
Joukahaisenkatu 3-5A, 20520 Turku, Finland
Sari Leppänen
Nokia Research Center, Computing Architectures Laboratory,
P.O. Box 407, 00045 Helsinki, Finland
Johan Lilius
Qaisar Ahmad Malik
Åbo Akademi University, Department of Information Technologies,
Joukahaisenkatu 3-5A, 20520 Turku, Finland

TUCS Technical Report
No 764, April 2006

Abstract

Telecommunication 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 and system fault tolerance should be ensured. Application of formal methods helps us to gain confidence in building correct software. However, to be used in practice, 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 and extends 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 the Lyra service specification, decomposition and distribution phases. Moreover, we extend Lyra to integrate reasoning about fault tolerance in the entire development flow.

Keywords: communicating systems, service-oriented development, fault tolerance, UML, B Method

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Modern telecommunication 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 telecommunication systems.

Formal methods have been traditionally used for reasoning about software correctness. Nevertheless, they are yet insufficiently well integrated into current development practice. Unlike formal methods, Unified Modeling Language (UML) [18] 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 a specific application domain.

Nokia Research Center has developed the design method Lyra [15] – a UML2-based service-oriented method specific to the domain of communicating systems and communication protocols. The design flow of Lyra is based on the 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 formalization of the Lyra development would allow us to ensure correctness of system design via automatic and formally verified construction. The achievement of such a formalization 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 [3] – 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.

To achieve system fault tolerance, we extend Lyra to integrate modelling of fault tolerance mechanisms into the entire development flow. We demonstrate how to formally specify error recovery by rollbacks as well as reason about error recovery termination.

The proposed formal specification and development patterns establish a background for automatic generation of formal specifications from UML2 models and expressing model transformations as refinement steps. Via automation of the UML2-based Lyra design flow we aim at smooth incorporation of formal methods into existing development practice.

2 Lyra: Service-Based Development of Communicating Systems

2.1 Overview of Lyra

Lyra [15] 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 pre-standardisation 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.

2.2 Lyra by Example

We model a positioning system which provides positioning services to calculate the physical location of a given item of user equipment (UE) in a mobile network [1, 2]. We consider Position Calculation Application Part (PCAP), which manages communication between two standard network elements. We assume that the PCAP functional requirements are correctly defined [1, 2] and, hence, focus on the architectural decomposition and distribution decisions.

The Service Specification phase starts from creating the domain model of the system. It describes the system with 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.1(a) and PSAP of the Positioning system – *I_User PSAP* is shown in Fig.1(b). The UML2 interfaces *I_ToPositioning* and *I_FromPositioning* define the signals and signal parameters of *I_user PSAP*. We formally describe the communication between a system level service and its user(s) in the *PSAPCommunication* state machine as illustrated in Fig.1(c). 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, one or more Reference Location Measurement Unit devices *(ReferenceLMU)* are contacted to provide additional measurements of radio signals. The radio measurements 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 produces the final estimation of the *UE* location. Let us observe that the 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 the service execution according to the identified stages.

In the Service Decomposition phase, we introduce the providers of external services into the domain model constructed previously, as shown in Fig.2(a). The model includes the external service providers *DB, UE, Ref-*

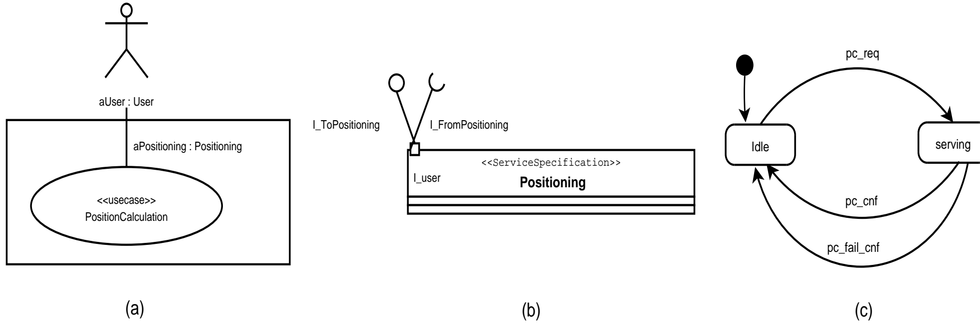


Figure 1: (a) Domain Model. (b) PSAP of Positioning. (c) State diagram.

erenceLMU and *Algorithm*, which are then defined as the 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 Service Access Points*, as presented in Fig.2(b).

Let us observe that the system 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 functionality related to a single execution stage or another logical piece of functionality.

In Fig.3(a) we present the architecture diagram of the *Positioning system*. Here *ServiceDirector* plays two roles: it controls the service execution flow and handles the communication on the PSAP. The behaviour of *ServiceDirector* is presented in Fig.3(b). 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 the specifications of 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 allocated on distant network elements.

In Fig.4(a) we illustrate the physical structure of the distributed positioning system. Here *Positioning_RNC* and *Positioning_SAS* represent the predefined network elements called RNC and SAS correspondingly. The standard interface *Iupc* is used in the communication between them. We map the functional architecture obtained at the previous stage to the given network architecture by distributing the service components between the network elements. The functional architecture of the *Positioning_SAS* network

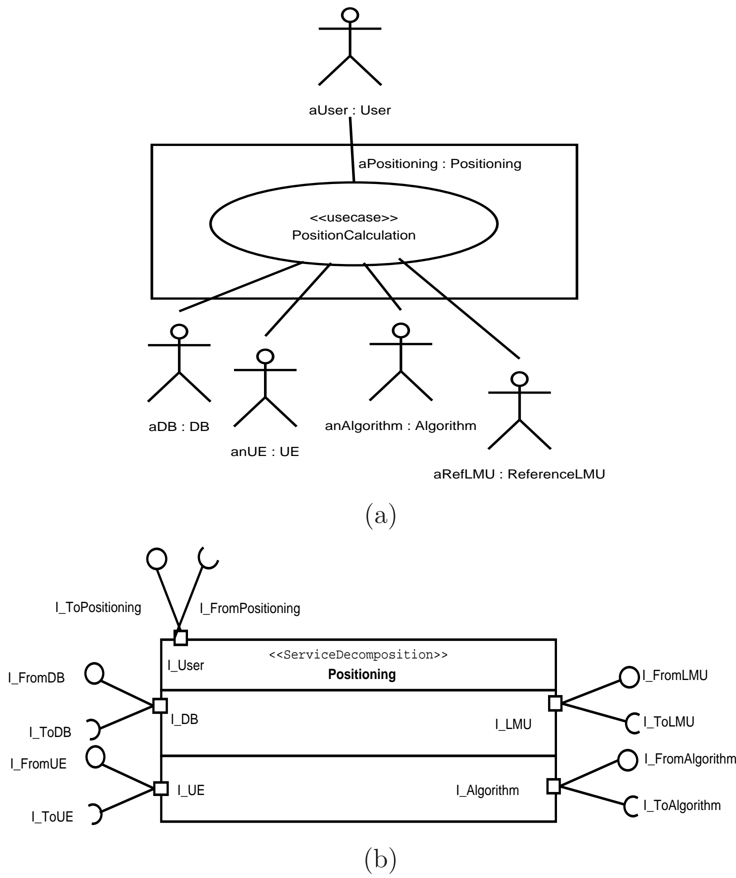
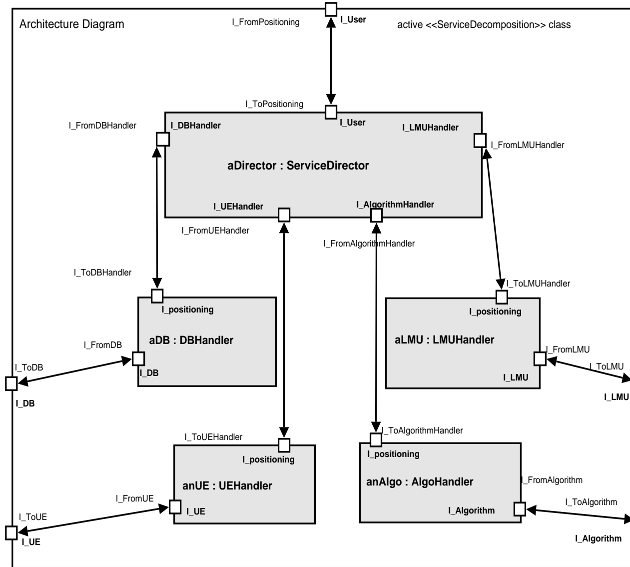


Figure 2: (a) Domain Model. (b) PSAP and USAPs of Positioning.

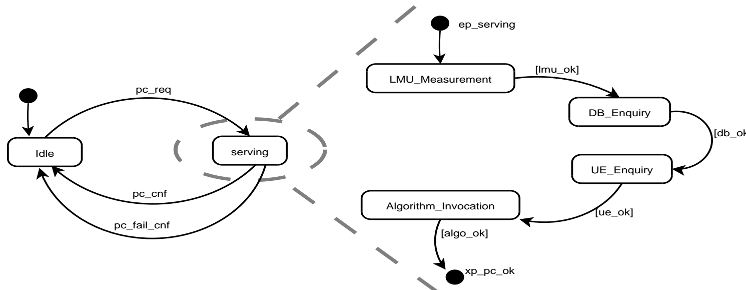
element is illustrated in Fig.4(b). The functionality of *ServiceDirector* specified at the Service Decomposition phase is now decomposed and distributed over the given network. *ServiceDirector_SAS* handles the interface towards the RNC network element and controls the execution flow of the positioning calculation process in the SAS network element.

Finally, at the *Service Implementation* phase we specify how the virtual 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 description of this stage can be found elsewhere [15, 1, 2].

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.



(a)



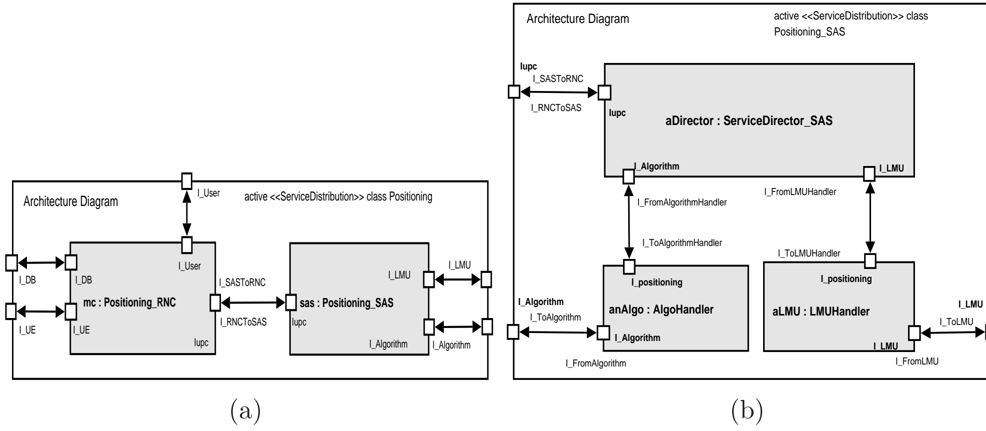
(b)

Figure 3: (a) PositionCalculation functional architecture. (b) Service Director: PSAP communication and execution control.

3 Developing Systems by Refinement in the B Method

The B Method [3] 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 [16]. Recently the B method has been extended by the Event B framework [4], which enables modelling of event-based systems. Event B is particularly suitable for developing distributed, parallel and reactive systems. In fact, this extension has incorporated the action system formalizm [6] in the B Method. In the rest of the paper, we refer to the B Method together with its extension Event B simply as B.

The tool support available for B provides us with the assistance for the



(a) (b)

Figure 4: (a) Architecture of service. (b) Architecture of Positioning_SAS.

entire development process. For instance, Atelier B [9], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. It has a plug-in for integrating modelling in Event B. Atelier B provides us with a high degree of automation in verifying correctness that improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

3.1 Modelling in B

The B Method adopts the top-down approach to system development. The development starts from creating a formal system specification. A formal specification is a mathematical model of the required behaviour of a system, or a part of a system. In B, a specification is represented by a collection of modules, called Abstract Machines. The Abstract Machine Notation (AMN), is used in constructing and verifying them. An abstract machine encapsulates a local state (local variables) of the machine and provides operations on the state. A simple abstract machine has the following general form:

MACHINE *AM*
SETS *TYPES*
VARIABLES v
INVARIANT I
INITIALISATION *INIT*
EVENTS
$E_1 \;=\; \ldots$
$\ldots$
$E_N \;=\; \ldots$
END

The machine is uniquely identified by its name *AM*. The state variables of the machine, v, are declared in the **VARIABLES** clause and initialised in $INIT$ as defined in the **INITIALISATION** clause. The variables in B are strongly typed by constraining predicates of the machine invariant I given in the **INVARIANT** clause. The invariant is usually defined as a conjunction of the constraining predicates and the predicates defining the properties of the system that should be preserved during system execution. All types in B are represented by non-empty sets. Local types can be introduced by enumerating the elements of the type, e.g., $TYPE = \{element1, element2, \ldots\}$, or by defining them as subsets of already existing types or sets.

The operations $E_1, \ldots, E_N$ of the machine are defined in the **EVENTS** clause. The operations are atomic meaning that, once an operation is chosen, its execution will run until completion without interference. There are two standard ways to describe an operation in B: either by the preconditioned operation **PRE** *cond* **THEN** *body* **END** or the guarded operation **SELECT** *cond* **THEN** *body* **END**. Here *cond* is a state predicate, and *body* is a B statement. If *cond* is satisfied, the behaviour of both the precondition operation and the guarded operation corresponds to the execution of their bodies. However, these operations behave differently when an attempt to execute them from a state where *cond* is false is undertaken. In this case the precondition operation leads to a crash (i.e., unpredictable or even non-terminating behaviour) of the system, while the execution of the guarded operation is blocked. The preconditioned operations are used to describe the operations that will be turned (implemented) into procedures called by the user. On the other hand, the guarded operations are useful when we have to specify system behaviour in terms of its reactions on the occurrence of certain events. The operations of event-based systems are often called *events*.

The B statements that we will use to describe the bodies of events have the following syntax:

$$
\begin{array}{lll}
S & == & x := e \mid \textbf{IF}\ cond\ \textbf{THEN}\ S1\ \textbf{ELSE}\ S2\ \textbf{END} \mid S1\ \textbf{;}\ S2 \mid \\
 & & x\ ::\ T \mid \textbf{ANY}\ z\ \textbf{WHERE}\ Q\ \textbf{THEN}\ S\ \textbf{END} \mid S1 \parallel S2 \mid \ \ldots
\end{array}
$$

The first three constructs - an assignment, a conditional statement and a sequential composition have the standard meaning. A sequential composition is disallowed in abstract specifications but permitted in refinements. 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. We use two kinds of nondeterministic statements – the nondeterministic assignment and the nondeterministic block. The nondeterministic assignment $x :: T$ assigns the variable x an arbitrary value from the given set (type) T. The nondeterministic block **ANY** z **WHERE** Q **THEN** S **END** introduces the new local variable z which is initialised (possibly nondeterministically) according to the predicate Q and then used in the statement S. Finally, $S1 \mid\mid S2$ models parallel (simultaneous) execution of $S1$ and $S2$ provided $S1$ and $S2$ do not have a conflict on state variables. The special case of the parallel execution is a multiple assignment, which is denoted as $x, y := e1, e2$.

The B statements are formally defined using the weakest precondition semantics [10]. Intuitively, for a given statement S and a postcondition P, the weakest precondition *wp(S,P)* describes the set of all such initial states from which execution of S is guaranteed to establish P. The weakest precondition semantics is a foundation for establishing correctness of specifications and verifying refinements between them. To show correctness (consistency) of an event-based system, we should demonstrate that its invariant is *true* in the initial state (i.e., after the initialisation is executed) and that every event preserves the invariant:

$$wp(INIT, I) = true, \quad \text{and}$$
$$g_i \wedge I \Rightarrow wp(E_i, I)$$

3.2 Refinement of Event-Based Systems

The basic idea underlying stepwise development in B is to design the system implementation gradually, by a number of correctness preserving steps called *refinements*. The refinement process starts from creating an abstract specification and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artefacts. In general, refinement process can be seen as a way to reduce nondeterminism of the abstract specification and replace abstract mathematical data structures by data structures implementable on a computer. Hence refinement allows us to introduce implementation decisions gradually.

Formally, we say that the statement S is refined by the statement S', written $S \sqsubseteq S'$, if, whenever S establishes a certain postcondition, so does S':

$$S \sqsubseteq S' \text{ if and only if for all postconditions } p: \; wp(S, p) \Rightarrow wp(S', p)$$

In the AMN the results of intermediate development stages – the refinement machines – have essentially the same structure as the more abstract specifications. In addition, a refinement machine explicitly states which specification it refines.

Assume that the refinement machine AM' given below is a result of refinement of the abstract machine AM:

REFINEMENT AM'
REFINES AM
VARIABLES v'
INVARIANT I'
INITIALISATION $INIT'$
EVENTS
$E_1 \ = \ \ldots$
$\ldots$
$E_N \ = \ \ldots$
END

In AM' we replace the abstract data structures of AM with concrete ones. The invariant of AM', I', defines now not only the invariant properties of the refined specification but also the connection between the newly introduced variables and the abstract variables that they replace. For a refinement step to be valid, every possible execution of the refined machine must correspond (via I') to some execution of the abstract machine. To demonstrate this, we should prove that $INIT'$ is a valid refinement of $INIT$, each event of AM' is a valid refinement of its counterpart in AM and that the refined specification does not introduce additional deadlocks, i.e.,

$$\begin{array}{l} wp(INIT',\ \neg wp(INIT, \neg I')) = true, \\ I \wedge I' \wedge g'_i \Rightarrow g_i \wedge wp(S', \neg wp(S, \neg I')), \quad \text{and} \\ I \wedge \bigvee_i^N g_i \Rightarrow g'_i \end{array}$$

Often refinement process introduces new variables and the corresponding computations (new events) on them, while leaving the previous variables and computations essentially unchanged. Such refinement is referred to as *superposition* refinement [7]. Let us consider the abstract machine AM_S and the refinement machine AM_SR:

MACHINE AM_S
VARIABLES a
INVARIANT I

INITIALISATION $INIT$
EVENTS
E = **WHEN** g
THEN S **END**

END

REFINEMENT AM_SR
REFINES AM_S
VARIABLES a, b
INVARIANT I'
VARIANT V
INITIALISATION $INIT'$
EVENTS
E = **WHEN** g
THEN S **END**
E_1 = **WHEN** g_1 **THEN** S_1 **END**
E_2 = **WHEN** g_2 **THEN** S_2 **END**
END

Observe that the refinement machine contains the new events E_1 and E_2 as well as the new clause **VARIANT**. The new events define computations on the newly introduced variables b and, hence, can be seen as the events refining the statement *skip* on the abstract variables. Every new event should decrease the value of the variant. This allows us to guarantee that new events cannot take the control forever, since the variant expression cannot be decreased infinitely. For each newly introduced event, we should demonstrate that the variant expression is a natural number and execution of the event decreases the variant, i.e.,

$$V \in NAT, \text{ and}$$
$$I' \wedge g_i \Rightarrow wp((n := V;\ S_i),\ n < V)$$

In B, there are also mechanisms for structuring the system architecture by modularisation. 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 incorporated 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.

Next we illustrate modelling and refinement in B by presenting a formal development of fault-tolerant communicating systems according to the Lyra methodology.

4 Towards Formalizing and Extending Lyra

4.1 Modelling a Service Component in B

In Section 2 we have defined 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 the providers of external services introduced at the Service Decomposition phase. However, the notion of a service component can be generalized to represent the service providers at different levels of abstraction. Indeed, even the entire *Positioning* system can be seen as a 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) that it is capable of providing. The *communicational* part is

an interface via which a service component receives requests to execute the service(s) and sends the results of service execution.

Execution of a service usually involves certain computations. We call the B representation of this part of a service component *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.

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, when modelling the communicational *(ACM)* part of *ACC*, we also specify how the service consumer places requests to execute a service in the operation *input* and reads the results of service execution in the operation *output*. The input parameters *param* and *time* of the operation *input* model the parameters of a request and the maximal time allowed for executing the service. For instance, in the Positioning System example described in Section 2, an arrival of the position calculation request – the signal *pc_req* – can be represented as an instantiation of the operation *input*. Moreover, the request might have parameters – the precision of position calculation defined by the service consumer and the maximal execution time defined by the system, e.g., according to the current network load. The parameters of the request are stored in the internal data buffer *in_data*, so they can be used by *ACAM* while performing the required computations.

In our initial specification we abstract away from the details of computations required to execute a service, i.e., *ACAM* is modelled as a statement non-deterministically generating the results of service execution. These results are stored in the internal output buffer *out_data*. The service consumer obtains the results of service provision as the output parameter *res* of the operation *output*. Already in the abstract specification we model possibility of failure – *out_data* might contain values representing the results of not only successful service executions but also failed ones. In our example, in case of successful execution, the signal *pc_cnf* together with the calculated position are sent to the service consumer. Otherwise, the signal *pc_fail_cnf* is generated.

While executing the operation *output*, the input and output buffers are emptied and the service component becomes ready to accept a new service request. Here we reserve the abstract constant *NIL* to model the absence of

data.

MACHINE *ACC*

SETS *DATA*

CONSTANTS *NIL*, *Abort_data*

PROPERTIES
$NIL \in DATA \wedge Abort_data \in DATA \wedge \neg\ (Abort_data = NIL)$

VARIABLES *in_data*, *out_data*

INVARIANT
$in_data \in DATA \wedge out_data \in DATA$

INITIALISATION
$in_data,\ out_data := NIL,\ NIL$

EVENTS

input(*param*,*time*) =
  PRE $param \in DATA \wedge time \in \mathbf{NAT1} \wedge \neg\ (param{=}NIL) \wedge in_data{=}NIL$
  THEN
    $in_data := param$
  END;

calculate =
  SELECT $\neg\ (in_data{=}NIL) \wedge out_data = NIL$
  THEN
    $out_data :\in DATA$ - $\{NIL\}$
  END;

$res \leftarrow output$ =
  PRE $\neg\ (out_data = NIL)$
  THEN
    $res := out_data\ ||$
    $in_data,out_data := NIL,\ NIL$
  END

END

In Lyra, a service component is usually 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 a service component according to the *ACC* pattern. Namely, the UML2 description of PSAP is translated into the communicational *(ACM)* part of the machine *ACC*. The functional *(ACAM)* part of *ACC* should be

instantiated by the data types specific to the modelled service component. This translation formalizes the *Service Specification* phase of Lyra.

Let us observe 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 a service component very abstractly. We have shown how it can be instantiated for the Positioning system example. While developing a more complex service component, this part can be instantiated with more elaborated data structures and the corresponding protocols for transferring them.

Next we discuss how to extend Lyra with the explicit representation of the fault tolerance mechanisms and then show the use of the ACC pattern in the entire Lyra development process.

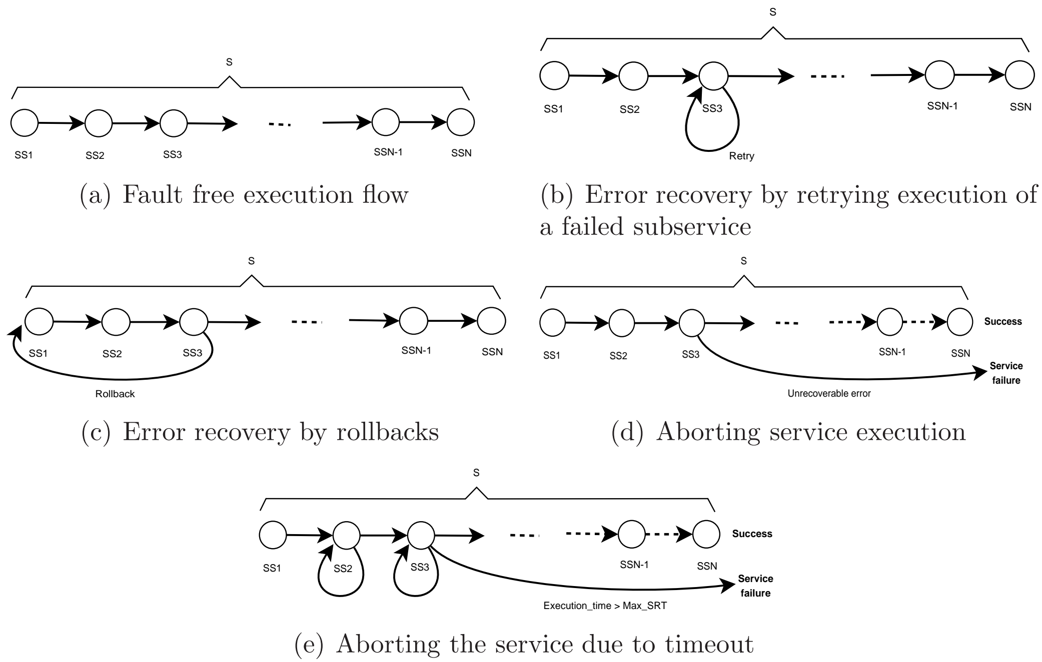
4.2 Introducing Fault Tolerance in the Lyra Development Flow

Currently the Lyra methodology addresses fault tolerance implicitly, i.e., by representating not only successful but also failed service provision in the Lyra UML models. However, it leaves aside modelling of mechanisms for detecting and recovering from errors – the fault tolerance mechanisms. We argue that, by integrating explicit representation of the means for fault tolerance into the entire development process, we establish a basis for constructing systems that are better resistant to errors, i.e., achieve better system dependability. Next we will discuss how to extend Lyra to integrate modelling of fault tolerance.

In the first development stage of Lyra we set a scene for reasoning about fault tolerance by modelling not only successful service provision but also service failure. In the next development stage – *Service Decomposition* – we elaborate on representation of the causes of service failures and the means for fault tolerance.

In the *Service Decomposition* phase we decompose the service provided by a service component into a number of stages (subservices). The service component can execute certain subservices itself as well as request other service components to do it. According to Lyra, the flow of the service execution is managed by a special service component called *Service Director*. It implements the behaviour of PSAP of a service component as specified earlier. Moreover, it co-ordinates the execution flow by enquiring the required subservices from the external service components.

In general, execution of any stage of a service can fail. In its turn, this might lead to failure of the entire service provision. Therefore, while specifying *Service Director*, we should ensure that it does not only orchestrates the fault-free execution flow but also handles erroneous situations. Indeed, as a result of requesting a particular subservice, *Service Director* can obtain



(a) Fault free execution flow

(b) Error recovery by retrying execution of a failed subservice

(c) Error recovery by rollbacks

(d) Aborting service execution

(e) Aborting the service due to timeout

Figure 5: Service decomposition: faults in the execution flow

a normal response containing the requested data or a notification about an error. As a reaction to the occurred error, *Service Director* might

- retry the execution of the failed subservice,
- repeat the execution of several previous subservices (i.e., roll back in the service execution flow) and then retry the failed subservice,
- abort the execution of the entire service.

The reaction of *Service Director* depends on the criticality of an occurred error: the more critical is the error, the larger part of the execution flow has to be involved in the error recovery. Moreover, the most critical (i.e., unrecoverable) errors lead to aborting the entire service. In Fig.5(a) we illustrate a fault free execution of the service S composed of subservices $S_1, \ldots, S_N$. Different error recovery mechanisms used in the presence of errors are shown in Fig.5(b) - 5(d).

Let us observe that each service should be provided within a certain finite period of time – the *maximal service response time Max_SRT*. In our model this time is passed as a parameter of the service request. Since each attempt of subservice execution takes some time, the service execution might be aborted even if only recoverable errors have occurred but the overall service execution time has already exceeded *Max_SRT*. Therefore, by introducing

Max_SRT in our model, we also guarantee termination of error recovery, i.e., disallow infinite retries and rollbacks, as shown in Fig.5(e).

Next we demonstrate how to represent the extended Lyra development as refinement in B.

5 Service-Oriented Development by Refinement in B

5.1 Formalizing Service Decomposition

In the first stage of our formalized development we used the UML2 models produced at the *Service Specification* phase to specify a service component according to the *ACC* pattern. The next step focuses on modelling the service execution flow with the incorporated fault tolerance mechanisms. Namely, we introduce a representation of *Service Director* into the abstract specification of a service component. This is done by refining the machine *ACC* to capture the design decisions made at *Service Decomposition* and *Service Distribution* phases. Hence, to derive the specification of *Service Director*, we use UML2 diagrams modelling both the functional and the platform-distributed architectures. In general, we should consider two cases:

1. *Service Director* is "centralized", i.e., it resides on a single network element,
2. *Service Director* is "distributed", i.e., different parts of the execution flow are orchestrated by distinct service directors residing on different network elements.

Assume for simplicity that the set of subservices required in the execution of the service S consists of three elements: S_1, S_2 and S_3. At the *Service Decomposition* phase, in both cases the model of the service component providing the service S looks as shown in Fig.6. The service distribution architecture diagram for the first case is given in Fig.7. In the second case, let us assume that the execution flow of the service component is orchestrated by two service directors: the *Service Director1*, which handles the communication on PSAP and communicates with the service component providing S_1, and *Service Director2*, which orchestrates the execution of the subservices S_2 and S_3. The service directors communicate with each other while passing the control over the corresponding parts of the execution flow. The architecture diagram depicting the overall arrangement for the second case is shown in Fig.9.

We model the decomposed service as a sequence over the abstract set *TASKS*. Each element of *TASKS* represents the individual subservice. Moreover, we introduce the abstract function *Next* to models the service execution

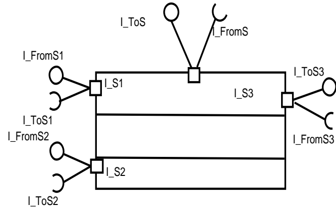


Figure 6: Service component with USAPs.

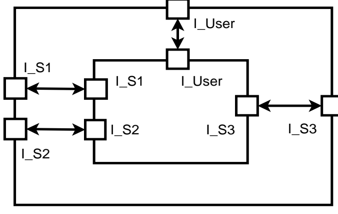


Figure 7: Architecture diagram (case 1)

flow. In case of the centralized *Service Director*, the subservices are executed one after another, i.e., the abstract representation of *Next* will be instantiated as follows:

$$Next(S_i) = S_{i+1}$$

for $i : 1..max_sv$, where max_sv is the maximal number of subservices required to execute the service.

In the second case, the function *Next* describes the execution flow from the point of view of the main service director, i.e., it treats the groups of services managed by other service directors as atomic steps in the execution flow. In our example, the service S_1 is managed by *Service Director1*, while S_2 and S_3 are managed by $ServiceDirector2$. In this case the function *Next* treats the execution of S_2 and S_3 as one execution step the performance of which is delegated to *Service Director2*. Hence, in this example *Next* will be instantiated as follows:

$$Next(S_1) = S_2, \quad \text{and} \quad Next(S_2) = S_4$$

The result of refinement of the machine ACC – the machine ACC_DEC – is given below.

REFINEMENT *ACC_DEC*
REFINES *ACC*
SETS
DATA; *TASK*; *RESPONSE* = {*OK*, *REPEAT*, *ROLLBACK*, *ABORT*}
CONSTANTS *Service*, *Eval*, *Next*, *max_sv*
PROPERTIES
$Service \in \mathbf{seq1}(TASK) \wedge \mathbf{size}(Service) = max_sv \wedge$
$Eval \in TASK \times DATA \rightarrow RESPONSE \wedge$
$\forall\ dd.\ (dd \in DATA \Rightarrow \neg\ (Eval(Service(1),dd) = ROLLBACK)) \wedge$
$Next \in 1\ ..\ max_sv \rightarrowtail\!\!\!\!\!\twoheadrightarrow 2\ ..\ max_sv{+}1 \wedge$
$1 \in \mathbf{dom}(Next) \wedge (max_sv{+}1) \in \mathbf{ran}(Next) \wedge$
$\forall ii.\ (ii \in \mathbf{dom}(Next) \wedge \neg\ (Next(ii){=}max_sv{+}1) \Rightarrow Next(ii) \in \mathbf{dom}(Next)) \wedge$

```
  ∀ii. (ii ∈ dom(Next) ⇒ ii<Next(ii)) ∧ ...
VARIABLES
  in_data, out_data, time_left, old_time_left,
  curr_task, resp, finished, results, curr_data
INVARIANT
  resp ∈ RESPONSE ∧
  results ∈ 1 .. max_sv ⇸ DATA-{NIL} ∧
  curr_data ∈ DATA ∧
  curr_task ∈ 1 .. max_sv+1 ∧
  (finished = FALSE ⇒ time_left>0) ∧
  time_left ≤ old_time_left ∧
  dom(results) ⊆ dom(Next) ∧
  (finished = TRUE ⇒ (resp=ABORT) ∨ (curr_task = max_sv+1)) ∧
  (finished = FALSE ⇒ curr_task ∈ 1 .. max_sv) ∧
  (finished = FALSE ⇒ curr_task ∈ dom(Next)) ∧
  (curr_task = max_sv+1 ⇒ ¬ (resp=ABORT)) ∧
  (finished = TRUE ∧ curr_task = max_sv+1 ⇒
  Next⁻¹ (curr_task) ∈ dom(results)) ∧ ...
INITIALISATION
  in_data, out_data := NIL, NIL ||
  time_left, old_time_left := max_time, max_time ||
  curr_task, resp := 1, OK ||
  finished, results := FALSE, ∅ ||
  curr_data := NIL
EVENTS

input(param,time) =
  PRE param ∈ DATA ∧ time ∈ NAT1 ∧ ¬ (param=NIL) ∧ in_data=NIL
  THEN
    in_data, time_left, old_time_left := param, time, time
  END;

handle =
  SELECT ¬ (in_data=NIL) ∧ finished = FALSE ∧ (time_left < old_time_left)
  THEN
    old_time_left := time_left; curr_data :∈ DATA-{NIL};
    resp := Eval(Service(curr_task),curr_data);
    CASE resp OF
      EITHER OK THEN
        results(curr_task) := curr_data;
        curr_task := Next(curr_task);
        IF curr_task = max_sv+1 THEN finished := TRUE END
```

```
      OR ROLLBACK THEN
        curr_task := Next^-1 (curr_task);
        results := {curr_task} ⩤ results
      OR REPEAT THEN skip
      OR ABORT THEN finished := TRUE
      END
    END
  END;

timer =
  SELECT ¬ (in_data=NIL) ∧ finished = FALSE ∧ (time_left = old_time_left)
  THEN
    CHOICE
      time_left :∈ {xx | xx ∈ NAT1 ∧ xx<time_left}
    OR
      time_left, resp := 0, ABORT ;
      finished := TRUE
    END
  END;

calculate =
  SELECT ¬ (in_data=NIL) ∧ out_data = NIL ∧ finished = TRUE
  THEN
    IF resp = ABORT THEN out_data := Abort_data
    ELSE
      out_data := results(Next^-1 (curr_task))
    END
  END;

res ← output =
  PRE ¬ (out_data = NIL)
  THEN
    res := out_data ; in_data,out_data := NIL, NIL
  END
END
```

The currently executed subservice is modelled by the variable *curr_task*. The results of the current subservice execution are stored in the variable *curr_data*. The results of all subservices already executed are accumulated in the variable *results*. The variable *finished* indicates the end of service execution. The variable is set to *TRUE* when the whole sequence of subservices has been executed or some unrecoverable error has occurred.

To model progress of time, we introduce the variable *time_left*. When a service request is received in the operation *input*, *time_left* is set to the maximal service response time *Max_SRT*, which is received as the second

parameter of *input*. The variable *old_time_left* is used to force interleaving between progress of the execution flow and the passage of time. The operation *timer* decreases the value of *time_left*, disables itself and enables the operation *handle*, which specifies the service co-ordinating behaviour of *Service Director*.

In the operation *handle*, we model not only requesting a certain subservice and obtaining its response, but also handling notifications about errors. We introduce the abstract function *Eval*, which evaluates the obtained response from a requested subservice. The result of evaluation is assigned to the variable *resp*. If the subservice was successfully executed then the variable *resp* gets the value *OK*. In this case the next element from the sequence of subservices is chosen for execution according to the function *Next*. If a benign failure has occurred and error recovery merely requires to retry the execution of the failed subservice then the variable *resp* is assigned the value *REPEAT*. This situation is illustrated in Fig. 5(b). However, if a more critical error has occurred, i.e., the variable *resp* gets the value *ROLLBACK*, the execution of several subservices preceding the failed service should be repeated as well. This case is depicted in Fig. 5(c). The inverse of the function *Next* defines which subservices should be re-executed, i.e., to which subservice the execution flow should rollback. In this case, we also delete the results of executing these subservices from *results*. Finally, if an unrecoverable error has occurred, i.e., the value of *resp* becomes *ABORT*, then the execution of the service is terminated (i.e., the variable *finished* is assigned *TRUE*) as shown in Fig. 5(d).

Let us note, that the variable *resp* also obtains the value *ABORT* once the timeout has occurred. This is modelled in the operation *timer*. The system might be in a state where the value of *time_left* had already became zero, while the execution of the service has not yet been finished, as depicted in Fig. 5(e).

In the refined machine *ACC_DEC* the guard of the event *calculate* is strengthened to ensure that the final result of the service is computed only after the execution of all subservices is finished (or aborted), i.e., when *finished = TRUE*.

The performed refinement has affected the *ACAM* part of the *ACC* pattern. The newly introduced events allowed us to define the details of execution of the decomposed service. In the **VARIANT** clause of *ACC_DEC* we not only ensure that the newly introduced events do not take control forever but also that execution of the service terminates.

Let us observe that our approach to introducing fault tolerance can be seen as an abstract implementation of the rollback error recovery frequently used in distributed systems [11]. Indeed, the operation *handle* defines the rollback procedure by co-ordinating error recovery according to the checkpoints defined by the function *Next*. The stable data storage is modelled by

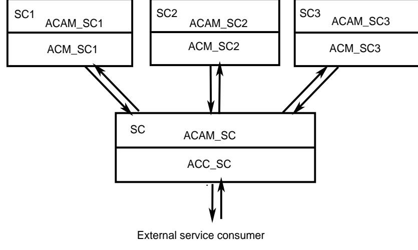


Figure 8: Specification architecture (case 1)

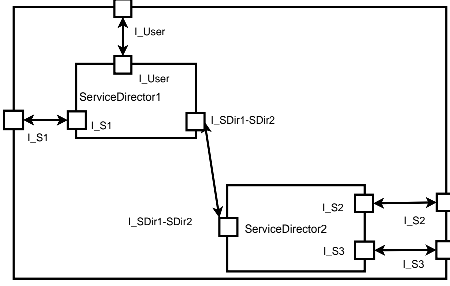


Figure 9: Architecture diagram (case 2)

the variable *results*. The operation *handle* ensures consistency of the system state by the appropriate updates of *results*.

While defining the execution flow over subservices in ACC_DEC, we abstracted away from modelling the details of the communication between *Service Director* and the external service providers – the USAP communication. Moreover, we omitted the explicit representation of the external service providers as such and modelled only the results of subservices provision. In our next refinement steps we decompose the obtained specification to introduce the detailed representation of the external service providers and the USAP communication.

5.2 Formal Modelling of Service Distribution

Let us first consider the case of a "centralized" service director shown in Fig. 7. It is easy to observe that the service component SC providing the service S plays a role of the service consumer for the service components $SC_1, \ldots, SC_N$ providing the subservices $S_1, \ldots, S_N$. We specify the service components $SC_1, \ldots, SC_N$ as the separate machines $ACC_SC1 \ldots$ ACC_SCN according to the proposed pattern ACC. The process of translating the UML2 models of $SC_1, \ldots, SC_N$ into B is similar to specifying SC at the *Service Specification* phase. The communicational parts of the included machines $ACC_SC1, \ldots, ACC_SCN$ specify the PSAPs of $SC_1, \ldots, SC_N$. 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 $SC_1, \ldots, SC_N$.

To define the mechanism for communicating with $SC_1, \ldots, SC_N$, we refine the operation *handle* to specify the communication on USAPs. Namely, we replace the nondeterministic assignments modelling specific stages of the service execution by the corresponding signalling scenarios: at the proper point of the execution flow, a desired service is requested by writing into the input channel of the corresponding included machine, and later the produced

results are read from the output channel of this machine. Graphically this arrangement is depicted in Fig.8.

Modelling the case of a distributed service director is more complex. Let us 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.9.

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*. When the result of *S1* is obtained, *ServiceDirector1* requests *ServiceDirector2* 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, *ServiceDirector1* 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*, coordinating 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* defined according to the *ACC_DEC* pattern by including into it the machines modelling the service components *SC1* and *SC2-SC3* and introducing the required communicating mechanisms. The result of this refinement step – the machine *SDirector1* – is given below (the parts of *SDirector1*, which coincide with the corresponding parts of *ACC_DEC* are replaced with dots).

REFINEMENT *SDirector1*
REFINES *ACC_DEC*
INCLUDES *Comp1*, *SDirector2*
CONSTANTS
  boolnum , . . .
PROPERTIES
$boolnum \in \mathbf{BOOL} \rightarrow 0\,..\,1 \wedge$
$boolnum(\mathbf{FALSE}) = 0 \wedge boolnum(\mathbf{TRUE}) = 1 \wedge$
$max_sv = 3 \wedge Next = \{1 \mapsto 2, 2 \mapsto 4\} \wedge$
$(Service = ([C1_Service] \frown SD2_Service))$
VARIABLES
  in_data, *out_data*, *time_left*, *old_time_left*,

```
  curr_task, resp, finished, results, curr_data, start_flag
INVARIANT
  start_flag ∈ BOOL
VARIANT
  boolnum(start_flag)
INITIALISATION
  ... || start_flag := TRUE
ASSERTIONS
  Next ∈ 1 .. max_sv ⤔ 2 .. max_sv+1 ∧
  1 ∈ dom(Next) ∧ (max_sv+1) ∈ ran(Next) ∧
   ∀ ii. (ii ∈ dom(Next) ⇒ ii<Next(ii)) ∧
   ∀ ii. (ii ∈ dom(Next) ∧ ¬ (Next(ii)=max_sv+1) ⇒ Next(ii) ∈ dom(Next))
EVENTS
input(param,time) = ... END;
handle =
  SELECT ¬ (in_data=NIL) ∧ finished = FALSE ∧
        (time_left < old_time_left) ∧
        ((curr_task=1 ∧ C1_out_data ≠ NIL) ∨
         (curr_task=2 ∧ SD2_out_data ≠ NIL))
  THEN
    old_time_left := time_left;
    CASE curr_task OF
      EITHER 1 THEN curr_data ← C1_output
      OR 2 THEN curr_data ← SD2_output
      END
    END;
    resp := Eval(Service(curr_task),curr_data);
    CASE resp OF
      EITHER OK THEN
        results(curr_task) := curr_data;
        curr_task := Next(curr_task);
        IF curr_task = max_sv+1 THEN finished := TRUE END
      OR ROLLBACK THEN
        curr_task := Next ⁻¹ (curr_task);
        results := {curr_task} ⩤ results
      OR REPEAT THEN skip
      OR ABORT THEN finished := TRUE
      END
    END;
    start_flag := TRUE
```

```
  END;
starter =
  SELECT ¬ (in_data=NIL) ∧ finished = FALSE ∧
        (time_left = old_time_left) ∧
         start_flag = TRUE ∧
       ((curr_task=1 ∧ C1_in_data = NIL) ∨
        (curr_task=2 ∧ SD2_in_data = NIL))
  THEN
    CASE curr_task OF
      EITHER 1 THEN C1_input(in_data,time_left)
      OR 2 THEN SD2_input(results(Next ^-1 (curr_task)),time_left)
      END
    END;
    start_flag := FALSE
  END;
timer = ... END;
calculate = ... END;
res ← output = ... END;
END
```

The machine *SDirector1* includes the machines *Comp1* and *SDirector2* specifying the service components *SC1* and *SC2-SC3* correspondingly. They are defined according to *ACC* and *ACC_DEC* patterns respectively. Since these machines can be obtained by a simple instantiation of these patterns, we omit their representation here.

The Service Director of *SC* communicates with the service component *SC1* and the Service Director of *SC2-SC3* by placing the corresponding requests in their input channels and reading the responses from their output channels. The order of requests is defined by the function *Next*. The function is instantiated in the **PROPERTIES** close to represent the particular architecture given in Fig. 9. Requesting the services from *CS1* and *SC2-SC3* is modelled in the operation *starter*, reading the output channels of *SC1* and *SC2-SC3* in the operation *handle*. Note, that the operation *handle* have been refined to explicitly model obtaining a response from the requested component.

In our consequent refinement step we focus on decomposition of *SC2-SC3*. 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.10. We omit the presentation of the detailed formal specifications – they are again obtained by recursive application of the proposed specification and refinement patterns.

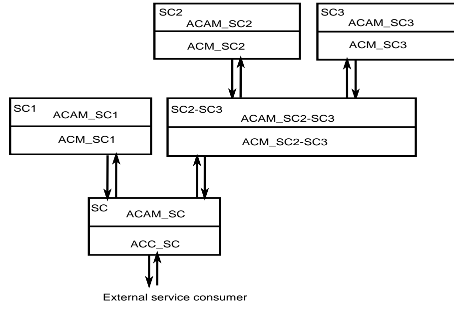


Figure 10: Specification architecture (case 2)

At the further 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 a 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.

5.3 Discussion

The proposed approach to formalizing Lyra in B allows us to verify correctness of the Lyra decomposition and distribution phases. This is done 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 formalization 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. We are planning to work on extending our B models to include parallel execution of services. 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.

Currently our approach can be implemented on a platform supporting the classical B Method and EventB. However, it can be adapted to the emerging RODIN platform [17] as well. The two major adjustments would need

to be done. Firstly, we would need to replace the preconditioned operations modelling communication between service components by the events, which explicitly work with input and output buffers of communicating components. Consequently, in the operation *handle* and *starter*, the calls of preconditioned operations would be replaced by the assignments to the corresponding buffers. Secondly, we would need to eliminate sequential composition and other control structures (like conditional and *CHOICE* statements) extensively used in our specifications. This can be achieved by splitting the operations using these control structures into the corresponding sets of events. Obviously, it would lead to rather artificial proliferation of new events. However, we believe that in the future the RODIN platform will allow us to conservatively extend the language and, hence, keep the used control structures.

6 Conclusions

In this paper we proposed a formal approach to development of communicating distributed systems. Our approach formalizes and extends Lyra [15] – the UML2-based design methodology adopted in Nokia. The formalization is done within the B Method [3] and its extension EventB [4] – 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 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 verified 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. Moreover, we demonstrated how to extend Lyra to integrate reasoning about fault tolerance in the entire development flow.

The initial formalization of Lyra has been undertaken using model checking techniques [15]. 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 B.

The pioneering work on formal development of distributed systems in Event B was done by Abrial et al. [5]. They demonstrated how to prove termination of a complex distributed protocol in Event B. In our work we use the principles defined in [5] to formalize the service-oriented development of complex communicating systems.

Yadav and Butler [22] used Event B to design fault tolerant transactions for replicated distributed database systems. They demonstrated how to formally verify by refinement that the design of a replicated database confirms to the one copy database abstraction. Similarly, in our work we use refinement to verify that the externally observable behaviour of distributed implementation of a service is equivalent to its centralized abstraction. However, our primary goal was not only formal verification of service development but also integration of modelling and refinement in B into the existing UML2-based development flow.

Treharne et al. [21] 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 the communicating system development.

Boström and Walden [8] 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 and 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 UML2.

There is active research going on translating UML to B [12, 13, 14, 19, 20]. Among these, the most notable is research conducted by Snook and Butler [19] 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 formalization of the entire UML-based development process in the domain of the communicating distributed systems. Moreover, we are planning to further enhance the proposed approach to address issues of concurrency.

Acknowledgments

This work is supported by IST FP6 RODIN Project.

References

[1] 3GPP. Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. Available at http://www.3gpp.org/ftp/Specs/html-info/25305.htm.

[2] 3GPP. Technical specification 25.453: UTRAN Iupc interface positioning calculation application part (pcap) signalling. Available at http://www.3gpp.org/ftp/Specs/html-info/25453.htm.

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

[4] J.-R. Abrial. Extending B without Changing it (for Developing Distributed Systems). *Proceedings of 1st Conference on the B Method*, pp.169-191, Springer-Verlag, November 1996, Nantes, France.

[5] J.-R. Abrial, D.Cansell, and D. Mery. A mechanically proved and Incremental development of IEEE 1394 Tree Identity Protocol. *Formal Aspects of Computing*, Vol.14, pp.215-227, 2003.

[6] R. Back. Refinement calculus, Part II: Parallel and reactive programs. *Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science*, Vol.430, pp.67-93, Springer-Verlag, 1990.

[7] R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3), pp.1-23, 1996.

[8] P. Boström and M. Walden. An Extension of Event B for Developing Grid Systems. In *Proceedings of ZB 2005:Formal Specification and Development in Z and B, Lecture Notes in Computer Science*, Vol.3455, pp.142-161, Springer-Verlag, Guildford, UK, 2005.

[9] Clearsy. *AtelierB: User and Reference Manuals.* Available at http://www.atelierb.societe.com/index_uk.html.

[10] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[11] E.N. Elnozahy, L. Alvisi, Y. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. *ACM Computing Surveys*, Vol.34, No.3, 2002.

[12] P. Facon, R. Laleau, H. Nguyen, and A. Mammar. Combining UML with the B Method for specification of database applications. Research report, CEDRIC laboratory, Paris, 1999.

[13] K. Lano, D. Clark, and K. Androutsopoulos. UML to B: Formal Verification of Object-Oriented Models. In *Proceedings of ICM, Lecture Notes in Computer Science*, Vol.2999, Springer.

[14] H. LeDang and J. Souquieres. Integrating UML and B specification techniques. In *Proceedings of Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, p.495, 2002.

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

[16] MATISSE. *Handbook for Correct System Construction*. Available at http://www.esil.univ-mrs.fr/ spc/matisse/Handbook/.

[17] Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.

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

[19] C. Snook and M. Butler. U2B - A tool for translating UML-B models into B. *UML-B Specification for Proven Embedded System Design*, Springer, 2004.

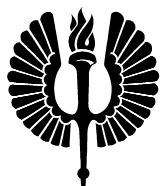
[20] C. Snook and M. Walden. Use of U2B for Specifying B Action Systems. In *Proceedings of International workshop on refinement of critical systems: methods, tools and experience (RCS'02)*, Grenoble, France, January 2002.

[21] H. Treharne, S. Schneider, and M. Bramble. Composing Specifications Using Communication. In *Proceedings of ZB 2003: Formal Specification and Development in Z and B, Lecture Notes in Computer Science*, Vol.2651, Springer, Turku, Finland, June 2003.

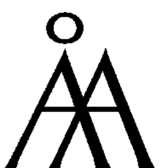
[22] D. Yadav and M. Butler. Application of Event B to Global Causal Ordering for Fault Tolerant Transactions. In *Proceedings of Workshop on Rigorous Engineering of Fault Tolerant Systems (REFT'2005)*, pp.93-102, Newcastle upon Tyne, UK, July 2005.

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 952-12-1716-2
ISSN 1239-1891