Modelling Fault Tolerance and Parallelism in Communicating Systems

Linas Laibinis[1], Elena Troubitsyna[1], and Sari Leppänen[2]

[1] Åbo Akademi University, Finland
[2] Nokia Research Center, Finland
{Linas.Laibinis, Elena.Troubitsyna}@abo.fi
Sari.Leppanen@nokia.com

Abstract. Telecommunication systems should have a high degree of availability, i.e., high probability of correct provision of requested services. To achieve this, correctness of software for such systems and system fault tolerance should be ensured. In this paper we show how to formalise and extend Lyra – a top-down service-oriented method for development of communicating systems. In particular, we focus on integration of fault tolerance mechanisms into the entire Lyra development flow.

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.

Nokia Research Center has developed the design method Lyra [6] – 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 organised 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 our previous work [5, 4] we proposed 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] – a formal refinement-based framework with automatic tool support. Moreover, to achieve system fault tolerance, we extended Lyra to integrate modelling of fault tolerance mechanisms into the entire development flow. We demonstrated how to formally specify error recovery by rollbacks as well as reason about error recovery termination.

In this paper we show how to extend our Lyra formalisation to model parallel execution of services. In particular, we demonstrate how such an extension affects the fault tolerance mechanisms incorporated into our formal models. The extension makes our formal models more complicated. However, it also gives us more flexibility in choosing possible recovery actions.

2 Previous Work

In this section we give a brief overview of on our previous results [5, 4] on formalising and verifying the Lyra development process. This work form the basis for new results presented in the next section.

2.1 Formalising Lyra

Lyra [6] is a model-driven and component-based design method for the development of communicating systems and communication protocols, developed in the Nokia Research Center. The method covers all industrial specification and design phases from pre-standardisation to final implementation.

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. 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. 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 integrated into the target environment. Examples of Lyra UML models from the Service Specification phase of a positioning system are shown on Fig.1.

To formalise the Lyra development process, we choose the B Method as our formal framework. The B Method [1] is an approach for the industrial development of highly dependable software. Recently the B method has been extended by the Event B framework [2, 7], which enables modelling of event-based systems. Event B is particularly suitable for developing distributed, parallel and reactive systems. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [3], 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 B Method adopts the top-down approach to system development. 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

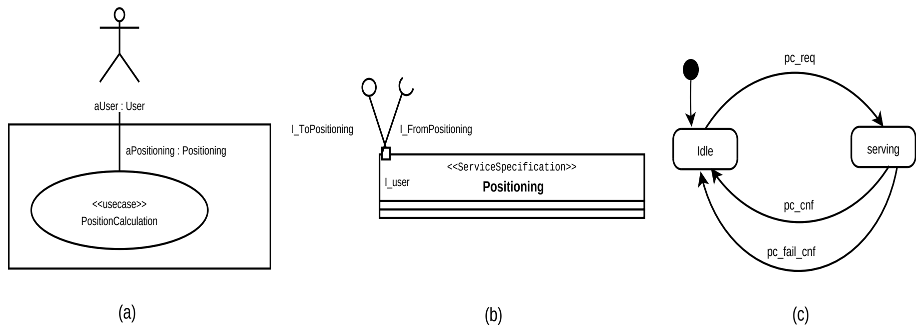


Fig. 1. (a) Domain Model. (b) Class Diagram of Positioning. (c) State Diagram.

and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artefacts.

While formalising Lyra, we single out a generic concept of a communicating service component and propose B patterns for specifying and refining it. In the refinement process a 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 network elements can also be captured by the notion of refinement. Below we present an excerpt from an abstract B specification pattern of a communicating service component.

The proposed approach to formalising Lyra in B allows us to verify correctness of the Lyra decomposition and distribution phases. 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.

```
MACHINE ACC
...
VARIABLES in_data, out_data, res
INVARIANT in_data ∈ DATA ∧ out_data ∈ DATA ∧ res ∈ DATA
INITIALISATION in_data, out_data, res := NIL, NIL, NIL
```

```
EVENTS
input =
  ANY param
  WHERE param ∈ DATA ∧ ¬(param = NIL) ∧ in_data = NIL
  THEN
    in_data := param
  END;

calculate =
  WHEN ¬(in_data = NIL) ∧ out_data = NIL
  THEN
    out_data :∈ DATA − {NIL}
  END;

output =
  WHEN ¬(out_data = NIL)
  THEN
    res := out_data ||
    in_data, out_data := NIL, NIL
  END
```

A B specification, called an *abstract machine*, encapsulates a local state (program variables) and provides operations on the state. In the Event B framework[1], such operations are called *events*. The events can be defined as

WHEN g **THEN** S **END**

or, in case of a parameterised event, as

ANY vl **WHERE** g **THEN** S **END**

where vl is a list of new local variables (parameters), g is a state predicate, and S is a B statement describing how the program state is affected by the event.

The events describe system reactions when the given **WHEN** or **WHERE** conditions are satisfied. The **INVARIANT** clause contains the properties of the system (expressed as predicates on the program state) that should be preserved during system execution. The data structures needed for specification of the system are defined in a separate module called *context*. For example, the abstract type $DATA$ and constant NIL used in the above specification are defined in the context ACC_Data, which can be accessed ("seen") by the abstract machine ACC.

[1] This work has been done using the Atelier B tool, supporting the Event B extension

The presented specification pattern is deliberatively made very simple. It describes a service component in a very abstract way – a service component simply receives some request data as the input, non-deterministically calculates non-empty result, which is then returned as the output. Using this specification as the starting point of our formal development gives us sufficient freedom to refine it into different kinds of service components. In particular, both the service components providing single services and the service components responsible for orchestrating service execution (called service directors) can be developed as refinements of the presented specification. Moreover, the defined specification and refinement patterns can be repeatedly used to gradually unfold the hierachical structure of service execution.

The proposed approach to formalising Lyra in B allows us to verify correctness of the Lyra decomposition and distribution phases. 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.

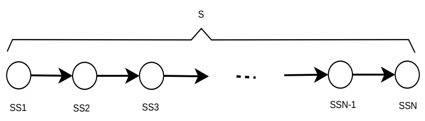
2.2 Introducing Fault Tolerance in the Lyra Development Flow

Currently the Lyra methodology addresses fault tolerance implicitly, i.e., by representing 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.

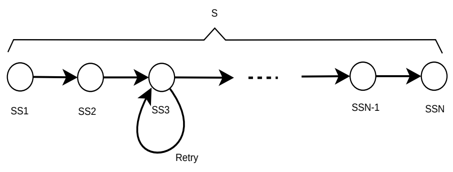
In the first development stage of Lyra we set a scene for reasoning about fault tolerance by modelling not only successful service provision but also service failure. In the next development stage – *Service Decomposition* – we elaborate on representation of the causes of service failures and the means 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*. *Service Director* co-ordinates the execution flow by enquiring the required subservices from the external service components.

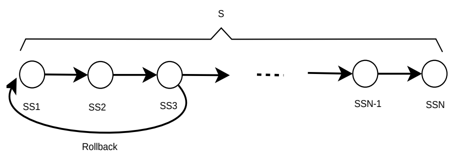
In general, execution of any stage of a service can fail. In its turn, this might lead to failure of the entire service provision. Therefore, while specifying *Service Director*, we should ensure that it does not only orchestrates the fault-free execution flow but also handles erroneous situations. Indeed,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



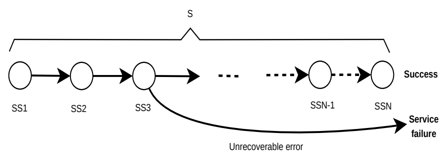
(a) Fault free execution flow



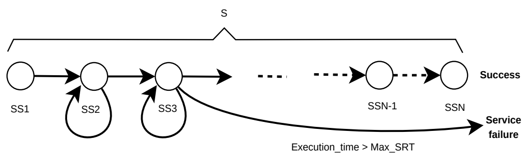
(b) Error recovery by retrying execution of a failed subservice



(c) Error recovery by rollbacks



(d) Aborting service execution



(e) Aborting the service due to timeout

Fig. 2. Service decomposition: faults in the execution flow

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.2(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.2(b) - 2(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.2(e).

3 Fault Tolerance in the Presence of Parallelism

Our formal model briefly described in the previous section assumes sequential execution of subservices. However, in practice, some of subservices can be executed in parallel. Such simultaneous service execution directly affects the fault tolerance mechanisms incorporated into our B models. As a result, they become more complicated. However, at the same time it provides additional, more flexible options for error recovery that can be attempted by *Service Director*.

3.1 Modelling Execution Flow

The information about all subservices and their required execution order becomes available at the Service Decomposition phase. This knowledge can be formalised as a data structure

$$Task\ :\ seq(\mathcal{P}(SERVICE))$$

Here $SERVICE$ is a set of all possible subservices. Hence, $Task$ is defined as a sequence of subsets of subservices. It basically describes the control flow for the top service in terms of required subservices. At the same time, it also indicates which subservices can be executed in parallel.

For example,

$$Task\ \ =\ <\{S1, S2\},\ \{S3, S4, S5\},\ \{S6\}>$$

defines the top service as a task that should start by executing the services $S1$ and $S2$ (possibly in parallel), then continuing by executing the services $S3$, $S4$, and $S5$ (simultaneously, if possible), and, finally, finishing the task by executing the service $S6$.

Essentially, the sequence $Task$ defines the data dependencies between subservices. Also, $Task$ can be considered as the most liberal (from point of view of parallel execution) model of service execution. In the Service Distribution phase the knowledge about the given network architecture becomes available. This can reduce the parallelism of service control flow by making certain services that can be executed in parallel to be executed in a particular order enforced by the provided architecture.

Therefore, $Task$ is basically the desired model of service execution that will serve as the reference point for our formal development. The actual service execution flow is modelled in by the sequence $Next$ which is of the same type as $Task$:

$$Next\ :\ seq(\mathcal{P}(SERVICE))$$

Since at the Service Decomposition phase we do not know anything about future service distribution, $Next$ is modelled as an abstract function (sequence), i.e., without giving its exact definition. However, it should be compatible with $Task$. More precisely, if $Task$ requires that certain services S_i and S_j should be executed in a particular order, this order should be preserved in the sequence

$Next$. However, $Next$ can split parallel execution of given services (allowed by $Task$) by sequentially executing them in any order.

So the sequence $Next$ abstractly models the actual control flow of the top service. It is fully defined (instantiated) only in the refinement step corresponding to the Service Distribution phase. For example, the following instantiation of $Next$ would be correct with respect to $Task$ defined above:

$$Next \ = \ < \{S2\},\ \{S1\},\ \{S4\},\ \{S3, S5\},\ \{S6\} >$$

Also, we have to take into account that *Service Director* itself can become distributed, i.e., different parts of service execution could be orchestrated by distinct service directors residing on different network elements. In that case, for every service director, there is a separate $Next$ sequence modelling the corresponding part of the service execution flow. All these control flows should complement each other and also be compatible with $Task$.

3.2 Modelling Recovery Actions

As we described before, a *Service Director* is the service component responsible for orchestrating service execution. It monitors execution of the activated subservices and attempts different possible recovery actions when these services fail. Obviously, introducing parallel execution of subservices (described in the previous subsection) directly affects the behaviour of *Service Director*.

Now, at each execution step in the service execution flow, several subservices can be activated and run simultaneously. *Service Director* should monitor their execution and react asynchronously whenever any of these services sends its response. This response can indicate either success or a failure of the corresponding subservice.

The formal model for fault tolerance presented in Section 2.2 is still valid. However, taking into account parallel execution of services presents *Service Director* with new options for its recovery actions. For example, getting response from one of active subservices may mean that some or all of the remaining active subservices should be stopped (i.e., interrupted). Also, some of the old recovery action (like retrying of service execution) are now parameterised with a set of subservices. The parameter indicates which subservices should be affected by the corresponding recovery actions.

Below we present the current full list of actions that *Service Director* may take after it receives and analyses the response from any of active subservices. Consequently, *Service Director* might

- **Continue** to the next service execution step. In case of successful termination of all involved subservices (complete success).
- **Wait** for response from the remaining active subservices. In case of successful termination of one of few active subservices (partial success).
- **Abort** the entire service and send the corresponding message to the user or requesting component. In case of an unrecoverable error or the service timeout.

– **Cancel** (a set of subservices) by sending the corresponding requests to interrupt their execution (partial abort). In case of a failure which requires to retry or rollback in the service execution flow.
– **Retry** (a set of subservices) by sending the corresponding requests to reexecute the corresponding subservices. In case of a recoverable failure.
– **Rollback** to a certain point of the service execution flow. In case of a recoverable failure.

Service Director makes its decision using special abstract functions needed for evaluating responses from service components. These functions should be supplied (instantiated) by the system developers at a certain point of system development.

Here is a small excerpt from the B specification of *Service Director* specifying the part where it evaluates a response and decides on the next step:

```
handle =
    ...
    resp := Eval(curr_task, curr_state);
    CASE resp OF EITHER
        CONTINUE THEN
            IF curr_task = size(Next) THEN finished := TRUE
            ELSE active_serv, curr_task := Next(curr_task + 1), curr_task + 1 END
        WAIT THEN skip
        RETRY THEN active_serv := active_serv ∪ Retry(curr_task, curr_state)
        CANCEL THEN active_serv := active_serv ∪ Cancel(curr_task, curr_state)
        ROLLBACK THEN curr_task := Rollback(...); active_serv := Next(curr_task)
        ABORT THEN finished := TRUE
    END
    ...
```

where the abstract functions Next, Retry, Cancel, and Rollback are defined (typed) as follows:

$\mathsf{Next} : \mathsf{seq}(\mathcal{P}(\mathsf{SERVICE}))$

$\mathsf{Eval} : 1..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \rightarrow \{\mathsf{CONTINUE}, \mathsf{WAIT}, \mathsf{RETRY}, \mathsf{CANCEL}, \mathsf{ROLLBACK}, \mathsf{ABORT}\}$

$\mathsf{Retry} : 1..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \nrightarrow \mathcal{P}(\mathsf{SERVICE})$

$\mathsf{Cancel} : 1..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \nrightarrow \mathcal{P}(\mathsf{SERVICE})$

$\mathsf{Rollback} : 2..\mathsf{size}(\mathsf{Next}) * \mathsf{STATE} \nrightarrow 1..\mathsf{size}(\mathsf{Next}) - 1$

4 Conclusions

In this paper we proposed a formal approach to development of communicating distributed systems. Our approach formalises and extends Lyra [6] – the UML2-based design methodology adopted in Nokia. The formalisation is done within the B Method [1] and its extension EventB [2] – a formal framework supporting system development by stepwise refinement. The proposed approach establishes a basis for automatic translation of UML2-based development of communicating systems into the refinement process in B. Such automation would enable smooth integration of formal methods into existing development practice.

In particular, in this paper we focused on integrating fault tolerance mechanisms into the formalised Lyra development process. A big challenge is formal modelling of parallel service execution and its effect on system fault tolerance. The ideas presented in this paper are implemented by extending our previously developed B models. The formalised Lyra development is verified by completely proving the corresponding B refinement steps using the Atelier B tool. At the moment, we are in the process of moving this development to new Event B language developed within the EU RODIN project [8].

Acknowledgements

This work has been supported by IST FP6 RODIN Project.

References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. 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.
3. Clearsy. *AtelierB: User and Reference Manuals.* Available at http://www.atelierb.societe.com/index_uk.html.
4. L. Laibinis, E. Troubitsyna, S. Leppänen, J.Lilius, and Q. Malik. Formal Service-Oriented Development of Fault Tolerant Communicating Systems. *Rigorous Development of Complex Fault-Tolerant Systems, Lecture Notes in Computer Science*, Vol.4157, chapter 14, pp.261-287, Springer-Verlag, 2006.
5. L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius, and Qaisar Malik. Formal Model-Driven Development of Communicating Systems. Proceedings of 7th International Conference on Formal Engineering Methods (ICFEM'05), LNCS 3785, Springer, November 2005.
6. S. Leppänen, M. Turunen, and I. Oliver. Application Driven Methodology for Development of Communicating Systems. *Forum on Specification and Design Languages*, Lille, France, 2004.
7. Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event B Language, online at http://rodin.cs.ncl.ac.uk/.
8. Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.