RODIN Deliverable D29

Final report on methodology

Authors:
M. J. Butler (Southampton University)
C. B. Jones (Newcastle University)
Abdolbaghi Rezazadeh (Southampton University)
Elena Troubitsyna (Aabo Akademi)

Public Document

30th October 2007

http://rodin.cs.ncl.ac.uk/
Abstract

One aim of the Rodin project is to contribute formal methods which will underpin the creation of fault-tolerant systems. This final report from WP2 (Methodology) points to the results achieved during the third year of the Rodin project and includes a bibliography of all of the relevant publications.
# Contents

1 Introduction ................................................. 5

2 The (generic) Event-B methodology .................. 6

3 Methods for Formal Development of Fault Tolerant Systems ...... 8
   3.1 Systems approach ........................................... 8
   3.2 Developing Fault Tolerant Control Systems ............... 9
      3.2.1 Systems Approach to Developing Control Systems ..... 9
      3.2.2 Refining Fault Tolerance Mechanism .................. 10
      3.2.3 Introducing Redundancy by Refinement ............... 12
      3.2.4 Summary of the Approach ............................. 13
   3.3 Methods of Ensuring Fault Tolerance in Service-Oriented Development ...... 14
      3.3.1 Systems Approach to Modelling a Service Component ... 14
      3.3.2 Introducing Fault Tolerance in the Lyra Development Flow .... 16
   3.4 Formal Approaches to Developing Fault Tolerant Agent Systems ........ 23
      3.4.1 CAMA: Basic Concepts ................................. 23
      3.4.2 Systems Approach to Specifying Mobile Agent Systems ... 24
      3.4.3 Introducing Scoping Mechanism ....................... 27
      3.4.4 Introducing Error Recovery by Refinement ............ 29
   3.5 Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems ..... 31
      3.5.1 Fault Tolerant Replication ............................ 31
      3.5.2 Systems Approach to Development of Replicated Database Systems ....... 32
   3.6 Discussion .................................................. 32

4 Other methodological issues .............................. 33
   4.1 Introduction ............................................... 33
   4.2 Requirement Modelling and Layering Specification ........... 33
   4.3 Complex Data Types and Layered Refinement ................ 34
   4.4 Proofs and Gluing Invariants Discovery .................... 35
   4.5 Adjusting Modelling Style to Achieve Higher Productivity .... 36
   4.6 Augmenting Semiformal Notation With Formal Notation ....... 37
   4.7 Verification Versus Proof-based Modelling .................. 37
   4.8 Conclusions ............................................... 38
5  **Specifics from the case studies**  
   5.1  Case Study 1: Formal Approaches in Protocol Engineering  
   5.2  Case Study 2: Engine Failure Management System  
   5.3  Case Study 3: MDA  
   5.4  Case Study 4: CDIS  
   5.5  Case Study 5: Ambient Campus  

6  **Other items of report**  
   6.1  Key meetings  
   6.2  Publications  

7  **The way forward**
Chapter 1

Introduction

Earlier reports (D9, D19) from RODIN’s WP2 have attempted to extract information about “methodology” from each of the case studies. This has led to questions and discussions about the use of aspects of the proposed methods within the case studies.

For this report, we have elected to emphasize use by pointing to the material in the reports on the relevant case studies — only the material in Chapters 2–4 is generic to the whole project.

Our work in preparing the DEPLOY IP proposal has also emphasized the need for any successful formal method to be integrated into the methods used by industrial partners. As such, we would now say that — while there is a core RODIN method – its benefits are manifested in five different deployments (see Sections 5.1–5.5).

The current report should be read in conjunction with the earlier D9 and D19 — we have not repeated the discussions there. Earlier references to methodology papers are, however, collected at the end of this report.

There is a concluding chapter (7) on the “challenges” we see for the future.
Chapter 2

The (generic) Event-B methodology

Jean-Raymond Abrial is close to finishing a new book that summarises the development method for Event-B. The title of the book is “Modeling in Event-B: System and Software Design” and it is to be published by Cambridge University Press. The book will contain approximately 16 chapters — the bulk of which are fully worked examples whose proofs have been constructed and/or checked with the RODIN tools. Furthermore, there will be a website that includes machine readable material including slides for teaching the material. This is an outstanding achievement in technology dissemination.

The book will be sent to the reviewers (for their personal use only!) but will not itself be a project deliverable for copyright reasons. It is however obvious that the dissemination of the RODIN results will be perfectly served by the publication of this book.

The emphasis throughout the book (and all RODIN methods) is on “correctness by construction” (CxC). This is in exact alignment with [LAB+06]. The overall plan of CxC is to begin with an abstract model and to introduce design decisions as refinements. At each stage of refinement, one proves that the previous specification will be met if the subsequent ones are fulfilled.

The examples in Jean-Raymond Abrial’s book show beautiful layering of design decisions. Of course, it is not always possible to get something right first time: one can end up in a blind alley. But anyone who claims to have a faultless method that ensures against mistaken design decisions is either a fool or a knave!

In addition to the material on abstraction and refinement, the book contains extremely useful advice on the structuring of requirements. There is also an emphasis on proving properties of (abstract) models as a way of increasing confidence that the system will meet the expectations of those who commission and/or use the system to be created. (RODIN tools also offer links to simulation.)

The main technical difference between Event-B and Abrial’s earlier “B book” [Abr96], is the use of guarded events. These can be used to provide more natural “models” (than in raw “B” or “VDM”) of many sorts of systems. Unfortunately, guarded events can also result in “deadlocks” and appropriate proof obligations have to be discharged to establish that this is not the case. (There should be no surprise about this: reactive systems can deadlock; for a simple enough system, the guards obviously cover all cases and the proof obligation is trivial to discharge.)

The new book also benefits enormously from the tool support offered by the RODIN Platform and Tools. The wide range of examples (roughly one per chapter) make it easier to relate
the method to new application areas. There is a chapter (number 9 in the current draft) on formal development of “sequential programs”; this is followed by chapters on both “concurrent programs” and (realisation as) “electronic circuits”.

Chapter 3

Methods for Formal Development of Fault Tolerant Systems

3.1 Systems approach

Within the RODIN project we worked on integrating formal methods and fault tolerance fields to facilitate the development of complex dependable systems. Techniques for achieving system fault tolerance provide us with the means to cope with failures of physical components and certain design mistakes. Meanwhile, formal methods provide us with the powerful design techniques enabling development of complex systems correct by construction. These techniques are based on abstraction, refinement and proofs.

The basic idea underlying formal stepwise development by refinement is to design the system implementation gradually by a number of correctness preserving steps, called refinements [BvW98]. The refinement process starts from creating an—abstract albeit unimplementable—specification and finishes with generating executable code. The intermediate stages yield specifications containing a mixture of abstract mathematical constructs and executable programming artefacts. In general, the refinement process can be seen as a way to reduce nondeterminism of the abstract specification, to replace abstract mathematical data structures by data structures implementable on a computer and to introduce underspecified (omitted) implementation decisions. Stepwise development allows us to better master complexity via abstraction, refinement and proof.

In the project we further developed systems approach to development of complex systems. The approach is exemplified by Butler et al. [BSS96]. The essence of system approach is that we start with an abstract formal model of the overall system, i.e., not only software but also the relevant part of the environment in which the system operates. While refining the abstract high-level model, we add the relevant details about environment behaviour as well as software implementation details. Finally, upon reaching the desirable level of details we decompose we overall specification, thus arriving at the low-level specification of software-implemented part as such.

While developing a system by an application of system approach we have a picture of whole system in mind. This facilitates better understanding of complex requirements that is perceived as the crucial point in developing modern computer-based systems. Indeed a faulty implementation of requirements can be easily diagnosed and corrected, while faulty design
resulting from incomplete or poorly defined requirements are expensive and difficult to fix.

The system approach proved to be especially suitable for developing fault tolerant systems. Fault tolerant systems contain mechanisms for detecting errors and recovering from them. The necessary condition for creating a methodology for developing such systems is to model mechanisms for error detection and recovery explicitly, as an intrinsic part of the specification. The systems approach allows us to achieve this. Indeed since the physical environment in which a system operates is explicitly specified, we can also model the effect which the error occurrence has on the environment behavior. In its turn this allows us to explicitly define error detection conditions and introduce error recovery procedures.

3.2 Developing Fault Tolerant Control Systems

3.2.1 Systems Approach to Developing Control Systems

Control systems constitute a large class of systems, many of which are safety-critical. Fault tolerance is directly contributing to system reliability – the ability of system operate correctly over given period of time under a given set of operating conditions. Obviously, reliability has a direct impact on system safety – freedom of accidents caused by the system.

In general, a control system is a reactive system with two main entities: a plant and a controller. The plant behaviour evolves according to the involved physical processes and the control signals provided by the controller. The controller monitors the behaviour of the plant and adjusts it to provide the intended functionality and maintain safety.

The properties of a control system we would like to enforce –such as safety and fault tolerance– cannot be attributed to either a plant or a controller alone. They are the properties of a system as a whole. Therefore, a systems approach provides us with a solid basis to design controllers for dependable systems. According to the systems approach, in our initial specification we model behaviour of the plant and the controller together.

The control systems are usually cyclic, i.e., at periodic intervals they get input from sensors, process it and output the new values to the actuators. In our abstract specification the sensors and actuators are represented by state variables shared by the plant and the controller. At each cycle, the plant reads the variables modelling actuators and assigns to the variables modelling the sensors. In contrast, the controller reads the variables modelling sensors and assigns the variables modelling the actuators. We assume that the reaction of the controller takes a negligible amount of time so the controller can react properly on changes of the plant state. The generic abstract specification is given in the machine ControlSystem below.

MACHINE ControlSystem
VARIABLES flag, state_variables
INVARIANT flag ∈ \{pl,contr,pred,det\} ∧ safety_inv ∧ fail ∈ BOOL
INITIALISATION flag := pl || fail := FALSE ||
EVENTS
Plant = WHEN flag=pl THEN evolution || flag := contr END;
Detection = WHEN flag=det THEN
    IF error is detected
    THEN fail := TRUE END || flag := det END;
Abort = WHEN flag=contr \(\land (not\;safe \lor fail = TRUE)\)
                     THEN shutdown END;
Control = WHEN flag=contr \(\land (safe \land fail = FALSE)\)
                     THEN control_action || flag := pred END;
Prediction = WHEN flag=pred THEN flag := pl END

END

The overall behaviour of the system is an alternation between the events modelling plant evolution and controller reaction. As a result of the initialisation, the plant operation becomes enabled. Once completed, the plant enables the controller. The behaviour of the controller follows the general pattern

\[\text{Error detection; Shutdown or Routine control; Prediction}\]

which is modelled by the corresponding assignments to variable flag.

The common mechanism for error detection is to find a discrepancy between the expected state of the system and the state which is observed in the reality. The event Prediction specifies the calculations required to predict the expected state. The event Detection models error detection by assigning value TRUE to variable fail. Due to high level of abstraction, in our initial specification the variable fail is assigned non-deterministically and the event Prediction is merely passing control to the plant.

Here we model failsafe - one of the most common mechanisms for error recovery. Failsafe error recovery is performed by forcing the system permanently to a safe though non-operation state (obviously this strategy is only appropriate where shutdown of the system is possible). In the initial specification we shut down the system if an error is detected or safety is breached (as defined in the condition of operation Abort). The shutdown is modelled as nondeterministic assignment to state variables.

Let us observe that system progress is stopped upon executing Abort event. Routine control operations can be executed provided the system is safe and fault-free. The routine control is specified by the event Control. We will decompose the overall system specification at the later refinement steps and eventually will arrive at the specification of the controller as such.

Our initial specification entirely defines the intended functionality of a fault-free system but leaves the means for fault tolerance underspecified. This is explained by the lack of the implementation details, which is typical at the early stages of development. These details become available at the later stages of the development, e.g., when results of hazard analysis conducted at a lower level are supplied. We complete specification of fault tolerance mechanisms by further refinement steps.

### 3.2.2 Refining Fault Tolerance Mechanism

We start refinement of control systems by replacing variable fail modelling error occurrence by the variables representing failures of system components. It is an example of data refinement. This data refinement expresses the fact that error occurs when one or several system components fail. The refinement relation defines the connection between the newly introduced variables and the variables that they replace.

While refining the specifications, we add the refinement relation to the invariant of the refined machine. In addition to replacing variable fail, our next refinement step also introduces
a more deterministic specification of plant behaviour. In the abstract specification we modelled plant behaviour as a nondeterministic update of sensor values. Such an abstraction includes modelling of both fault-free and faulty behaviour. In the refined specification of the plant we separate them.

The behaviour of fault-free plant evolves according to the certain physical laws which can be expressed as the corresponding mathematical functions. Moreover, these functions can be further adjusted to model deviations caused by imprecision of sensors measuring the physical values. We use these functions to model fault-free behaviour of the plant. On the other hand, faults make the plant to deviate from the dynamics defined by these functions. We specify occurrence of faults non-deterministically and locally to the plant. However, we model the effect of fault occurrence by assigning sensors values different from the ones which they would obtain in the absence of faults.

Refinement of plant behaviour also allows us to refine error detection mechanism. Observe that the mathematical functions modelling the plant's behaviour as described above can be used to predict the next state of the system. Hence we can refine operation Prediction to include the calculation of the expected system states. Furthermore, we also can refine operation Detection to check whether the expected state matches the obtained sensor readings. The detected mismatch signals the presence of an error. The result of this refinement step is a specification of the following form:

```
MACHINE ControlSystemRefined
REFINES ControlSystem
VARIABLES flag, state, variables of ControlSystem
    new variables for modelling failures of components
    variables modelling expected states
INVARINT constraints of variables AND data refinement relation
INITIALISATION initialization of variables
EVENTS
    Plant = WHEN flag=pl THEN
        simulation of evolution of the plant based on
        the corresponding physical laws and
        non-deterministic occurrence of failures ; flag := contr END;
    Detection = WHEN flag=det THEN
        IF real state does not match expected state
        THEN failures of components are detected END;
        flag := det END;
    Abort = WHEN flag=contr AND (not refined_safe OR components failed)
        THEN shutdown END;
    Control = WHEN flag=contr AND
        (refined_safe AND components are fault-free)
        THEN control_action ; flag := pred END;
    Prediction = WHEN flag=pred
        THEN calculate next expected state
        using the same physical laws as for simulating the plant ;
        flag := pl END
END
```
3.2.3 Introducing Redundancy by Refinement

Let us note that in the specification obtained at the previous refinement step all errors are considered to be equally critical, i.e., leading to the shutdown. While introducing redundancy at our next refinement step, we obtain a possibility to distinguish between criticality of errors. The distinction is done with respect to safety as follows: an occurrence of a marginal error means that the system can continue its functioning without compromising safety; on the other hand, an occurrence of critical errors endangers system safety so the shutdown needs to be executed.

In terms of the state space, we split the set of faulty system states into the subset of faulty but safe (and hence operational) states and the subset of faulty and unsafe states. The introduction of redundancy into the specification allows us to transform the system from fault nonmasking to masking. Fault masking is a mechanism that reconfigures the system upon error detection in such a way that the effect of the error is nullified. For instance, the simplest and the most common implementation of fault masking is triple modular redundancy (TMR) arrangement.

TMR uses majority voting to single out a failed component and reconfigures the outputs of redundant components in such a way that the erroneous output is disregarded. An occurrence of errors that can be masked leaves both functioning of the system and safety intact, so we call these errors marginal. On the other hand, occurrence of critical errors, i.e., the errors that cannot be masked, jeopardizes normal functioning and safety of the system. Hence the partitioning of faulty system states can also be thought of as the partitioning into a subset of masked and unmasked errors. The introduction of redundancy as described above results in the refinement machine of the form ControlSystemRefinedRedundant.

Observe that, although the guard of Abort event is unchanged by this refinement step, the event nevertheless becomes enabled less often. This is because Detection event is now distinguishes between the critical and marginal errors so that the occurrence of the marginal failures does not enable Abort anymore.

MACHINE ControlSystemRefinedRedundant
REFINES ControlSystemRefined
VARIABLES flag, state_variables of ControlSystem
    new variables for modelling redundancy
    variables modelling expected states
    variables for modelling failures of components
INVARIANT constraints of variables AND data refinement relation
INITIALISATION initialization of variables
EVENTS
  Plant = WHEN flag=pl THEN
    simulation of the behaviour of the plant
    with redundant components
    and the occurrence of component failures; flag := contr END;
  Detection = WHEN flag=det THEN
    IF mismatch between the states of redundant components is detected
    THEN
IF the error cannot be masked
THEN critical failure of redundant components is detected END
ELSIF the real state does not match the expected state
THEN critical failure of other components is detected
ELSE flag := det END . . . END;

Abort = WHEN flag=contr AND (not refined_safe OR components failed)
THEN shutdown END;

Control = WHEN flag=contr AND
          (refined_safe AND components are fault-free OR failures are masked)
THEN control_action ; flag := pred END;

Prediction = WHEN flag=pred
THEN calculate next expected state using
            the same physical laws as for simulating the plant;
            flag := pl END

END

At the final refinement step the system is decomposed into the plant and the controller. The specification of the controller includes all events except Plant. From this specification we can generate the executable code.

3.2.4 Summary of the Approach

According to our approach the system model evolves as follows:

• Abstract specification of entire system: the initial specification captures requirements for routine control, models failure occurrence and defines safety property as a part of its invariant

• Specification with refined error detection mechanism: the abstract specification is augmented with the representation of failures of the components, a more elaborated description of plant’s dynamics and a detailed description of error detection.

• Specification of the system supplemented with redundancy: the specification is refined to describe the behaviour of the redundant components and the control over them. The error detection mechanism is enhanced to distinguish between criticality of failures.

• Decomposition: the specification of overall system is split into specifications of the controller and the plant.

• Implementation: executable code of controller is produced.

This approach was described in detail by Laibinis and Troubitsyna [LT04]. The approach was further developed by Ilic et al. [IT05, ITLS06a] to include treatment of transient faults.
3.3 Methods of Ensuring Fault Tolerance in Service-Oriented Development

3.3.1 Systems Approach to Modelling a Service Component

The service-oriented development is becoming a popular approach in the development of complex computer-based systems. The notion of service provides a convenient mechanism for modelling and reasoning about interactions and hence, service-oriented development paradigm is particularly suitable for the development of communicating systems. Within Nokia Research Centre, service-oriented development paradigm has been conceptualized within Lyra framework [LTO04].

Lyra is 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.

Within RODIN we formalized Lyra in Event-B [LTL+06b] and introduced reasoning about fault tolerance into the design flow [LTL+06a]. One of the central concepts of Lyra is the notion of a service component. It is defined as a coherent piece of functionality that provides its services to a service consumer via Provided Service Access Point PSAP(s). The notion of a service component can be generalized to represent the service providers at different levels of abstraction.

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 systems 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 defined by the events input and output, we also specify how the service consumer places requests to execute a service in the event input and reads the results of service execution in the event output, as defined in the machine ACC below. The parameters of the request in_data, can be used by ACAM while performing the required computations, which produces the result out_data.

MACHINE ACC

SEES Data
VARIABLES

in_data
out_data
res

INVARIANTS

inv1 : in_data ∈ DATA
inv2 : out_data ∈ DATA
inv3 : res ∈ DATA

EVENTS

INITIALISATION

BEGIN
  act1 : in_data, out_data, res := NIL, NIL, NIL
END

EVENT input

ANY
  param
WHERE
    grd1 : param ∈ DATA ∧ ¬ (param = NIL)
THEN
  act1 : in_data := param
END

EVENT calculate

WHEN
  grd1 : ¬ (in_data = NIL)
THEN
  act1 : out_data ∈ DATA \ {NIL}
END

EVENT output

WHEN
  grd1 : ¬ (out_data = NIL)
THEN
  act1 : res := out_data
  act2 : in_data, out_data := NIL, NIL
END

END
In our initial specification we abstract away from the details of computations required to execute a service, i.e., ACAM (the event calculate) is modelled as a statement non-deterministically generating the results of service execution. These results are stored in the output buffer out_data. The service consumer obtains the results of service provision as a result of executing event 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.

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.

The specification ACC defines a generic specification patterns which is recursively used to define service components on different levels of abstraction throughout the entire development flow. The pattern 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 [LTL+06b]. While developing a more complex service component, this part can be instantiated with more elaborated data structures and the corresponding protocols for transferring them.

### 3.3.2 Introducing Fault Tolerance in the Lyra Development Flow

Originally, the Lyra methodology addressed fault tolerance implicitly, i.e., by representing not only successful but also failed service provision in the Lyra UML models. However, it left aside modelling of mechanisms for detecting and recovering from errors – the fault tolerance mechanisms. We demonstrated that, by integrating explicit representation of the means for fault tolerance into the entire development process, we established a basis for constructing systems that are better resistant to errors, i.e., achieve better system dependability.

In Service specification phase a service is specified according to ACC pattern, i.e., is essentially modelled by its interface. In the Service Decomposition and Service Distribution phases we decompose the service provided by a service component into a number of subservices and map then onto a given network architecture. 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 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.3.1(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.3.1(b) - 3.1(d).

Let us observe that each service should be provided within a certain finite period of time – the maximal service response time \( \text{Max}_\text{SRT} \). 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 \( \text{Max}_\text{SRT} \). Therefore, by introducing \( \text{Max}_\text{SRT} \) in our model, we also guarantee termination of error recovery, i.e., disallow infinite retries and rollbacks, as shown in Fig.3.1(e).

A service director implementing fault tolerance mechanisms can be defined as the refinement of ACC pattern, as shown in the machine ServiceDirector below.

MACHINE ServiceDirector

REFINES ACC
SEES Data2

VARIABLES

\[ \text{in}\_\text{data} \]
\[ \text{out}\_\text{data} \]
\[ \text{res} \]
\[ \text{time}\_\text{left} \]
\[ \text{old}\_\text{time}\_\text{left} \]
\[ \text{curr}\_\text{task} \]
\[ \text{finished} \]
\[ \text{results} \]
\[ \text{curr}\_\text{state} \]
\[ \text{active} \]
\[ \text{changed} \]
\[ \text{resp} \]

INVARIANTS

\[ \text{inv1} : \text{time}\_\text{left} \in \mathbb{N} \land \text{old}\_\text{time}\_\text{left} \in \mathbb{N} \land \text{curr}\_\text{task} \in \mathbb{N} \]
\[ \text{inv2} : \text{finished} \in \text{BOOL} \]
\[ \text{inv9} : \text{results} \in 0..(\text{curr}\_\text{task} - 1) \rightarrow \text{STATE} \]
\[ \text{inv3} : \text{curr}\_\text{state} \in \text{STATE} \land \text{curr}\_\text{task} \in 0..\text{max}\_\text{next} \land \text{active} \in \mathcal{P}(\text{SERVICE}) \]
\[ \text{inv4} : \text{changed} \in \text{BOOL} \land (\text{finished} = \text{FALSE} \Rightarrow \text{time}\_\text{left} > 0) \]
\[ \text{inv5} : \text{time}\_\text{left} \leq \text{old}\_\text{time}\_\text{left} \]
\[ \text{inv6} : \text{resp} \in \text{RESPONSE} \]
\[ \text{inv7} : \text{finished} = \text{TRUE} \land \neg (\text{resp} = \text{ABORT}) \Rightarrow \text{curr}\_\text{task} = \text{max}\_\text{next} \]
\[ \text{inv8} : \text{resp} = \text{ABORT} \Rightarrow \text{finished} = \text{TRUE} \]
\[ \text{inv10} : \text{finished} = \text{TRUE} \Rightarrow \text{resp} \in \{\text{SUCCESS}, \text{ABORT}\} \]

EVENTS

INITIALISATION

BEGIN
\[ \text{act1} : \text{in}\_\text{data}, \text{out}\_\text{data}, \text{res} := \text{NIL}, \text{NIL}, \text{NIL} \]
\[ \text{act2} : \text{time}\_\text{left}, \text{old}\_\text{time}\_\text{left}, \text{curr}\_\text{task} := \text{max}\_\text{time}, \text{max}\_\text{time}, 0 \]
\[ \text{act3} : \text{resp}, \text{finished} := \text{SUCCESS}, \text{FALSE} \]
\[ \text{act8} : \text{results} := \emptyset \]
\[ \text{act4} : \text{curr}\_\text{state} := \text{STATE} \]
\[ \text{act5} : \text{active}, \text{changed} := \emptyset, \text{FALSE} \]
END

EVENT input
REFINES input
ANY
\[
\begin{align*}
\text{param} & \quad \text{time} \\
\text{mode} & \\
\text{WHERE} & \\
\text{grd}_1 & : \text{param} \in \text{DATA} \land \text{time} \in \mathcal{N}_1 \land \text{mode} \in \text{MODE} \land \neg (\text{param} = \text{NIL}) \\
\text{THEN} & \\
\text{act}_1 & : \text{in}_{-\text{data}}, \text{curr}_{-\text{state}} := \text{param}, \text{Init}_{-\text{state}}(\text{param} \mapsto \text{mode}) \\
\text{act}_2 & : \text{results}, \text{curr}_{-\text{task}} := \emptyset, 0 \\
\text{act}_3 & : \text{finished}, \text{changed} := \text{FALSE}, \text{TRUE} \\
\text{act}_4 & : \text{time}_{-\text{left}}, \text{old}_{-\text{time}_{-\text{left}}} := \text{time}, \text{time} \\
\text{act}_5 & : \text{resp} := \text{SUCCESS} \\
\text{END} & \\
\text{EVENT} & \text{timer} \\
\text{WHEN} & \\
\text{grd}_1 & : \neg (\text{in}_{-\text{data}} = \text{NIL}) \land \text{finished} = \text{FALSE} \\
\text{grd}_2 & : \text{changed} = \text{TRUE} \land \text{time}_{-\text{left}} = \text{old}_{-\text{time}_{-\text{left}}} \\
\text{grd}_3 & : \{tt \mid tt \in \mathcal{N}_1 \land tt < \text{time}_{-\text{left}} \} \neq \emptyset \\
\text{THEN} & \\
\text{act}_1 & : \text{time}_{-\text{left}} \in \{tt \mid tt \in \mathcal{N}_1 \land tt < \text{time}_{-\text{left}} \} \\
\text{END} & \\
\text{EVENT} & \text{timer}_{-\text{out}} \\
\text{WHEN} & \\
\text{grd}_1 & : \neg (\text{in}_{-\text{data}} = \text{NIL}) \land \text{finished} = \text{FALSE} \\
\text{grd}_2 & : \text{changed} = \text{TRUE} \land \text{time}_{-\text{left}} = \text{old}_{-\text{time}_{-\text{left}}} \\
\text{grd}_3 & : \{tt \mid tt \in \mathcal{N}_1 \land tt < \text{time}_{-\text{left}} \} = \emptyset \\
\text{THEN} & \\
\text{act}_1 & : \text{time}_{-\text{left}}, \text{resp} := 0, \text{ABORT} \\
\text{act}_2 & : \text{finished} := \text{TRUE} \\
\text{END} & \\
\text{EVENT} & \text{handle}_{-\text{SUCCESS}} \\
\text{WHEN} & \\
\text{grd}_1 & : \neg (\text{in}_{-\text{data}} = \text{NIL}) \land \text{finished} = \text{FALSE} \land \text{changed} = \text{TRUE} \\
\text{grd}_2 & : \text{Eval}(\text{curr}_{-\text{task}} \mapsto \text{curr}_{-\text{state}}) = \text{SUCCESS} \\
\text{grd}_3 & : \text{curr}_{-\text{task}} < \text{max}_{-\text{next}} \land \text{time}_{-\text{left}} < \text{old}_{-\text{time}_{-\text{left}}} \\
\text{THEN} & \\
\text{act}_1 & : \text{results}, \text{curr}_{-\text{task}} := \text{results} \cup \{\text{curr}_{-\text{task}} \mapsto \text{curr}_{-\text{state}}\}, \text{curr}_{-\text{task}} + 1 \\
\text{act}_2 & : \text{active} := \text{Next}(\text{curr}_{-\text{task}} + 1) \\
\text{act}_3 & : \text{old}_{-\text{time}_{-\text{left}}} := \text{time}_{-\text{left}} \\
\text{act}_4 & : \text{resp} := \text{SUCCESS} \\
\text{END} & \\
\text{EVENT} & \text{handle}_{-\text{SUCCESS}_{-\text{complete}}} \\
\text{WHEN} & \\
\end{align*}
\]
\text{grd1} : \neg (\text{in\_data} = \text{NIL}) \land \text{finished} = \text{FALSE} \land \text{changed} = \text{TRUE} \\
\text{grd2} : \text{Eval}(\text{curr\_task} \mapsto \text{curr\_state}) = \text{SUCCESS} \land \text{curr\_task} = \text{max\_next} \\
\text{grd3} : \text{time\_left} < \text{old\_time\_left} \\
\text{THEN} \\
\text{act1} : \text{finished} := \text{TRUE} \\
\text{act2} : \text{old\_time\_left} := \text{time\_left} \\
\text{act3} : \text{resp} := \text{SUCCESS} \\
\text{END} \\
\text{EVENT handle\_REPEAT} \\
\text{WHEN} \\
\text{grd1} : \neg (\text{in\_data} = \text{NIL}) \land \text{finished} = \text{FALSE} \land \text{changed} = \text{TRUE} \\
\text{grd2} : \text{Eval}(\text{curr\_task} \mapsto \text{curr\_state}) = \text{REPEAT} \\
\text{grd3} : \text{time\_left} < \text{old\_time\_left} \\
\text{THEN} \\
\text{act1} : \text{active} := \text{active} \cup \text{Repeat}(\text{curr\_task} \mapsto \text{curr\_state}) \\
\text{act2} : \text{old\_time\_left} := \text{time\_left} \\
\text{act3} : \text{resp} := \text{REPEAT} \\
\text{END} \\
\text{EVENT handle\_CANCEL} \\
\text{WHEN} \\
\text{grd1} : \neg (\text{in\_data} = \text{NIL}) \land \text{finished} = \text{FALSE} \land \text{changed} = \text{TRUE} \\
\text{grd2} : \text{Eval}(\text{curr\_task} \mapsto \text{curr\_state}) = \text{CANCEL} \\
\text{grd3} : \text{time\_left} < \text{old\_time\_left} \\
\text{THEN} \\
\text{act1} : \text{active} := \text{active} \cup \text{Cancel}(\text{curr\_task} \mapsto \text{curr\_state}) \\
\text{act2} : \text{old\_time\_left} := \text{time\_left} \\
\text{act3} : \text{resp} := \text{CANCEL} \\
\text{END} \\
\text{EVENT handle\_CONTINUE} \\
\text{WHEN} \\
\text{grd1} : \neg (\text{in\_data} = \text{NIL}) \land \text{finished} = \text{FALSE} \land \text{changed} = \text{TRUE} \\
\text{grd2} : \text{Eval}(\text{curr\_task} \mapsto \text{curr\_state}) = \text{CONTINUE} \\
\text{grd3} : \text{time\_left} < \text{old\_time\_left} \\
\text{THEN} \\
\text{act2} : \text{old\_time\_left} := \text{time\_left} \\
\text{act3} : \text{resp} := \text{CONTINUE} \\
\text{END} \\
\text{EVENT handle\_ROLLBACK} \\
\text{WHEN} \\
\text{grd1} : \neg (\text{in\_data} = \text{NIL}) \land \text{finished} = \text{FALSE} \land \text{changed} = \text{TRUE} \\
\text{grd2} : \text{Eval}(\text{curr\_task} \mapsto \text{curr\_state}) = \text{ROLLBACK} \\
\text{grd5} : \text{time\_left} < \text{old\_time\_left}
THEN
act1 : curr_task := Rollback(curr_task ↦ curr_state)
act2 : results := (0 .. Rollback(curr_task ↦ curr_state) − 1) < results
act3 : active := Next(Rollback(curr_task ↦ curr_state))
act4 : old_time_left := time_left
act5 : resp := ROLLBACK
END

EVENT handle_ABORT
WHEN
grd1 : ¬ (in_data = NIL) ∧ finished = FALSE ∧ changed = TRUE
grd2 : Eval(curr_task ↦ curr_state) = ABORT
grd3 : time_left < old_time_left
THEN
act1 : finished := TRUE
act2 : old_time_left := time_left
act3 : resp := ABORT
END

EVENT calculate_ABORT
REFINES calculate
WHEN
grd1 : ¬ (in_data = NIL) ∧ finished = TRUE ∧ resp = ABORT
THEN
act1 : out_data := Abort_data
END

EVENT calculate_ELSE
REFINES calculate
WHEN
grd1 : ¬ (in_data = NIL) ∧ finished = TRUE ∧ ¬ (resp = ABORT)
THEN
act1 : out_data := Output(results(curr_task − 1))
END

EVENT output
REFINES output
WHEN
grd1 : ¬ (out_data = NIL)
THEN
act1 : res := out_data
act2 : in_data, out_data := NIL, NIL
END

EVENT read_response
ANY
WHERE
  grd3 : ss ∈ active \ data ∈ DATA \ \{NIL\}
  grd1 : changed = FALSE
  grd2 : active \\neq \emptyset

THEN
  act1 : curr_state := update(ss \mapsto curr_state \mapsto data)
  act2 : active := active \setminus \{ss\}
  act3 : changed := TRUE

END

END

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 flow.

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, time_left is set to the maximal service response time Max_SRT. The variable old_time_left is used to force interleaving between progress of the execution flow and the passage of time. The event timer decreases the value of time_left, disables itself and enables the event group handle, which specifies the service co-ordinating behaviour of Service Director.

In the event group 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. 3.1(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. 3.1(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. 3.1(d).

Let us note, that the variable resp also obtains the value ABORT once the timeout has occurred. This is modelled in the event timer. The system might be in a state where the
value of \textit{time\_left} had already became zero, while the execution of the service has not yet been finished, as depicted in Fig. 3.1(e).

While defining the execution flow over subservices in \textit{ServiceDirector}, we abstracted away from modelling the details of the communication between \textit{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.

Essence the fault tolerance mechanisms is preserved in the further refinement steps, so we omit their representation here. However, in the case distribution of services over the given architecture involves communication between several service directors, the model of fault tolerance mechanism becomes significantly more complex. This is also the case for the model which explicitly deals with parallel execution of services. More details on these issues can be found elsewhere [LTL07].

3.4 Formal Approaches to Developing Fault Tolerant Agent Systems

3.4.1 CAMA: Basic Concepts

Agent technology offers a number of advantages over traditional distributed systems, such as asynchronous communication, anonymity of individual agents and ability to change operational context. Middleware is the core part of agent systems providing coordination and mobility mechanisms. In RODIN we used systems approach to develop fault tolerant agent systems.

The work was carried out within CAMA (Context Aware Mobile Agent) systems framework [AIR06]. The CAMA inter-agent communication is based on the LINDA paradigm [D.G85], which provides a set of language-independent coordination primitives that can be used for co-ordination of several independent agents. These primitives allow agents to put tuples (vectors of values) in a shared tuple space, remove them from it, and test the shared space for their presence.

The major contribution of CAMA is a novel mechanism to structure inter-agent communication, allowing groups of communicating agents to work in isolated subspaces called \textit{scopes}. A scope is a dynamic container for tuples. The tuples contained within a scope are visible only to the agents participating in this particular scope. Hence, a scope provides an isolated coordination space for its agents.

Each agent carries special attributes describing the functionality it implements. Such an abstract description of agent functionality is called \textit{role}. Graphically a scope with agents participating in it is depicted in Fig.3.2. Let us observe, that each agent participating in the scope \(S\) should implement at least one role from the set \textit{Roles\_S} of the roles supported by \(S\).

Scopes also act as the service discovery mechanism. Agents can look up for activities or services by analysing scope attributes. Scope attributes are represented as a LINDA tuple and the discovery procedure is based on LINDA tuple matching.

In its turn, a \textit{location} is a container for scopes. Locations constitute the core part of a CAMA system. Finally, \textit{platform} provides an execution environment for agents. It is composed of a virtual machine for code execution, networking support, and client middleware for interacting with a location.
A graphical representation of an overall CAMA system is given in Fig. 2. As it is easy to see, the middleware supporting agent execution is distributed between agents and locations. In the project we formally developed a part of CAMA middleware supporting activities of an agent in a location, i.e., the parts of middleware denotes as $A-L$ and $L-A$ in Fig. 3.3. The development was done by an application of systems approach.

**3.4.2 Systems Approach to Specifying Mobile Agent Systems**

A typical behaviour of an agent can be described as follows: an agent connects to a location and then joins an existing scope. In a scope it can cooperate with agents participating in the same scope. When an agent leaves the scope, it either joins another scope or disconnects from the location. To support this behaviour the CAMA middleware provides three categories of operations: location engagement, scoping mechanism, and communication. The location engagement operations associate or disassociate an agent with a location. The scoping mechanism operations allow an agent to enquiry for available scopes, create new scopes, destroy previously created scopes, join and leave existing scopes.

One of the major challenges in designing agent systems lies in ensuring interoperability of agents. This problem can only be properly addressed if we define the essential properties of the overall agent system, derive the properties to be satisfied by a location and each agent, and ensure that they are preserved in the agent and location development. This goal can be achieved by adopting the system approach to developing agent systems, i.e., modelling the entire set of agents together with a location.

Below we present specification and refinement of middleware supporting activities of agents on a single location. Our development starts from an abstract specification given in the machine
Cama, which models the entire agent system, i.e., the agents and the location together. The variable \textit{agents} represents the set of agents that joined the location. The operations \textit{Engage} and \textit{Disengage} model joining and leaving the location correspondingly. While an agent is in the location, it performs some computations, as modelled by the operation \textit{NormalActivity}. To express that these computations are performed locally within the agent and hence do not affect the abstract state of the system, we model them by the statement \textit{skip}.

\textbf{MACHINE Cama}\\
\textbf{SETS Agents}\\
\textbf{VARIABLES agents}\\
\textbf{INVARIANT} agents \subseteq Agents\\
\textbf{INITIALISATION} agents := \emptyset\\
\textbf{EVENTS}\\
\textit{Engage} = \text{ANY} \text{ \textit{aa} WHERE } \text{\textit{aa} } \in \text{Agents} \land \text{\textit{aa} } \notin \text{agents}\\
\text{THEN} agents := agents \cup \{\text{\textit{aa}}\} \text{ END};\\
\textit{NormalActivity} = \text{ANY} \text{ \textit{aa} WHERE } \text{\textit{aa} } \in \text{Agents} \land \text{\textit{aa} } \in \text{agents}\\
\text{THEN skip END ;}\\
\textit{Disengage} = \text{ANY} \text{ \textit{aa} WHERE } \text{\textit{aa} } \in \text{Agents} \land \text{\textit{aa} } \in \text{agents}\\
\text{THEN agents := agents - \{\text{\textit{aa}}\} END}

In our initial specification we abstracted away from explicit modelling of the system behaviour in the presence of faults, e.g., due to temporal losses of connection. Although, the result of failure – disengagement of an agent from the location – is implicitly modelled in the operation \textit{Disengage}. In our first refinement step we introduce an explicit representation of the system behaviour in the presence of temporal losses of connection.

Let us observe that in most cases an agent loses connection only for a short period of time. After connection is restored, the agent is willing to continue its activities virtually uninterrupted. Therefore, after detecting a connection loss, the location should not immediately disengage the disconnected agent but rather set a deadline before which the agent should reconnect. If the disconnected agent restores its connection before the deadline then it can continue its normal activity. However, if the agent fails to do it, the location should disengage the agent.

Such a behaviour can be adequately modelled by the timeout mechanism. Upon detecting a disconnection the location activates a timer. If the agent reconnects before the timeout then the timer is stopped. Otherwise, the location forcefully disengages the disconnected agent. To model this behaviour, in the first refinement step we introduce the variable \textit{timers} representing the subset of agents that have disconnected but for which the timeouts have not expired yet. Moreover, we introduce the variable \textit{ex_agents} to model the subset of agents that missed their reconnection deadline and should be disengaged from the location. Finally, we add the new events \textit{Disconnect}, \textit{Connect} and \textit{Timer} to model agent disconnection, reconnection and timeout correspondingly.

To ensure that the refined system does not introduce additional deadlocks, we define the variant, which constraints the number of successive disconnections and reconnections. The constant \textit{Disconn_limit} defines the maximal number of successive disconnections. The variable \textit{disconn_limit} obtains the value \textit{Disconn_limit} in the initialisation. Each newly introduced events decreases the value of the variant either by decreasing the value of \textit{disconn_limit} (when
an agent disconnects) or by removing elements from the set timers (when a disconnected agent either reconnects or misses the reconnection deadline). The value of the variant is restored by executing the NormalActivity event.

In our specification we assume that an agent failure due to the loss of connection is detected by the location. However, an agent might by itself detect an error in its functioning and leave the location. Therefore, the agent might get disengaged from the location due to the following three reasons:

- because it has successfully completed its activities in the location,
- due to the disconnection timeout,
- due to a spontaneous failure detected by the agent itself.

In the refined specification given in the machine Cama1 below, we model all these different types of leaving by splitting the operation Disengage into three corresponding operations: NormalLeaving, TimerExpiration and AgentFailure.

REFINEMENT Cama1
REFINES Cama
CONSTANTS Disconn_limit
PROPERTIES Disconn_limit ∈ NAT ∧ Disconn_limit > 1
INITIALISATION
agents := ∅ || timers := ∅ ||
ex_agents := ∅ || disconn_limit := Disconn_limit
VARIABLES agents, timers, ex_agents, disconn_limit
INVARIANT
agents ⊆ timers ⊆ agents ∧
timers ∩ ex_agents = ∅ ∧
disconn_limit ∈ NAT
VARIANT
\text{card(timers)} + 2^{\text{disconn\_limit}}
EVENTS
Engage = ANY aa WHERE aa ∈ Agents ∧ aa ∉ agents
\hspace{1em} \text{THEN \text{agents := agents \cup \{aa\} END;}}
NormalActivity = ANY aa WHERE aa ∈ agents
\hspace{1em} \text{THEN \text{disconn\_limit := Disconn\_limit END;}}
NormalLeaving ref Disengage = ANY aa WHERE
\hspace{1em} (aa ∈ agents) ∧ (aa ∈ timers) ∧ (aa ∉ ex_agents)
\hspace{1em} \text{THEN \text{agents := agents - \{aa\} END;}}
TimerExpiration ref Disengage = ANY aa WHERE
\hspace{1em} (aa ∈ agents) ∧ (aa ∈ ex_agents)
\hspace{1em} \text{THEN \text{agents := agents - \{aa\} \| \text{ex_agents := ex_agents - \{aa\} END;}}
AgentFailure ref Disengage = ANY aa WHERE
\hspace{1em} (aa ∈ agents) ∧ (aa ∈ timers) ∧ (aa ∉ ex_agents)
\hspace{1em} \text{THEN \text{agents := agents - \{aa\} END;
Connect = ANY aa WHERE (aa ∈ agents) ∧ (aa ∈ timers)
    THEN timers := timers - {aa} END;

Disconnect = ANY aa WHERE
    (aa ∈ agents) ∧ (aa ∉ ex_agents) ∧ (aa ∉ timers) ∧ disconn_limit > 1
    THEN timers := timers ∪ {aa} || disconn_limit := disconn_limit - 1 END;

Timer = ANY aa WHERE (aa ∈ agents) ∧ (aa ∈ timers)
    THEN ex_agents := ex_agents ∪ {aa} || timers := timers - {aa} END

END

The refined specification Cama1 is a result of superposition refinement and atomicity refinement of the abstract specification Cama. This refinement step allowed us to introduce both error detection and error recovery into the system specification. Hence, already at a high level of abstraction we specify fault tolerance as an intrinsic part of the system behaviour.

3.4.3 Introducing Scoping Mechanism

In the abstract specification and the first refinement step we mainly focused on modelling interactions of agents with the location. Our next refinement step introduces an abstract representation of the scopes as an essential mechanism that governs agent interactions while they are involved in cooperative activities.

The creation of a scope is initiated by an agent, which consequently becomes the scope owner. The other agents might join the scope and become engaged into the scope activities. The agents might also leave the scope at any instance of time. The scope owner cannot leave the scope but might close it (this action is not permitted for other agents). When the scope owner closes the scope, it forces all agents participating in the scope to leave.

The introduction of the scoping mechanism also enforces certain actions to be executed when an agent decides to leave a location. Namely, to leave the location an agent should first leave or close (if it is the scope owner) all the scopes in which it is active.

The scoping mechanism has deep impact on modelling error recovery in agent systems. For instance, if the scope owner irrecoverably fails, then, to recover the system from this error, the location should close the affected scope and force all agents in this scope to leave.

We refine the machine Cama2 to specify the scoping mechanism described above. In the refinement machine Cama2, we introduce the variable scopes, which is defined as a relation associating the active scopes with the agents participating in them. Moreover, we add the variable sowner to model scope owners. It is defined as a total function from the active scopes to agents.

We define the new events Create, Join, Leave and Delete to model creating a scope by the owner, joining and leaving it by agents, as well as closing a scope. In the excerpt from the refinement machine Cama2, we demonstrate the newly introduced variables and events as well as the effect of the refinement on the events AgentFailure and TimerExpiration. The guard of the event NormalLeaving is now strengthened to disallow an agent to leave the location when it is still active in some scopes.

REFINEMENT Cama2
REFINES Cama1
SETS ScopeName
CONSTANTS ScopeLimit

PROPERTIES ScopeLimit ∈ NAT1

DEFINITIONS activeAgent(aa) == (aa \notin ex_agents \land aa \notin timers)

VARIABLES . . . , scopes, sowner, slimit

INVARIANT

scopes ∈ ScopeName \leftrightarrow agents \land sowner ∈ ScopeName \rightarrow agents \land

\text{dom}(sowner) = \text{dom}(scopes) \land sowner \subseteq scopes \land slimit ∈ NAT

VARIANT slimit

EVENTS

Create = ANY aa, nn WHERE

(aa ∈ agents) \land (activeAgent(aa)) \land

(nn ∈ ScopeName) \land (nn \notin \text{dom}(scopes)) \land slimit > 0

THEN

\text{CHOICE}

scopes, sowner := scopes \cup \{nn \mapsto aa\}, sowner \cup \{nn \mapsto aa\}

\text{OR skip END} ||

slimit := slimit - 1

END;

Join = ANY aa, nn WHERE

(aa ∈ agents) \land (activeAgent(aa)) \land

(nn ∈ \text{dom}(scopes)) \land ((nn \mapsto aa) \notin scopes) \land slimit > 0

THEN

\text{CHOICE}

scopes := scopes \cup \{nn \mapsto aa\}

\text{OR skip END} ||

slimit := slimit - 1

END;

Leave = ANY aa, nn WHERE

nn ∈ \text{dom}(scopes) \land aa ∈ agents \land aa \neq sowner(nn) \land

activeAgent(aa) \land (nn \mapsto aa) \in scopes \land slimit > 0

THEN

scopes := scopes - \{nn \mapsto aa\} ||

slimit := slimit - 1

END;

Delete = ANY aa, nn WHERE

nn ∈ \text{dom}(scopes) \land aa ∈ agents \land

activeAgent(aa) \land aa=sowner(nn) \land slimit > 0

THEN

scopes, sowner := \{nn\} \leftarrow scopes, \{nn\} \leftarrow sowner ||

slimit := slimit - 1

END;

NormalLeaving = ANY aa WHERE

aa ∈ agents \land aa \notin timers \land

activeAgent(aa) \land aa \notin \text{ran}(scopes) \land aa \notin \text{ran}(sowner)

THEN agents := agents - \{aa\} END;
TimerExpiration = ANY aa WHERE aa ∈ agents ∧ aa ∈ ex_agents

    THEN
        agents := agents - {aa}; scopes := scopes \[\{aa\}\];
        scopes := sowner \[-1\] [[{aa}]] ! scopes;
        ex_agents := ex_agents - {aa}; sowner := sowner \[\{aa\}\]
    END;

AgentFailure = ANY aa WHERE aa ∈ agents ∧ activeAgent(aa)

    THEN
        agents := agents - {aa}; scopes := scopes \[\{aa\}\];
        scopes := sowner \[-1\] [[{aa}]] ! scopes; sowner := sowner \[\{aa\}\]
    END

Let us observe that an agent does not always successfully creates or joins a scope. This is modelled by the skip statements in bodies of operations Create and Join. At the later refinement steps we will elaborate on the causes of these failures.

Termination of the added new events Create, Join, Leave and Delete is guaranteed by introducing the new variable slimit, which serves as the variant expression for the new event operations. As in the previous refinement step, the value of the variant expression is reset in the NormalActivity event.

This refinement step is again an example of a superposition refinement. The newly defined variables and events allowed us to introduce the general representation of the scoping mechanism.

3.4.4 Introducing Error Recovery by Refinement

In our current specification the event AgentFailure treats any agent failure as an irrecoverable error. Indeed, upon detecting an error, the failed agent is removed from the scope in which it participates and then disengaged from the location. However, usually upon detecting an error the agent at first tries to recover from it (possibly involving some other agents in the error recovery). If the error recovery eventually succeeds, then the normal operational state of the agent is restored. Otherwise, the error is treated as irrecoverable.

In our next refinement step, we introduce error recovery into our specification. We define the the variable astate to model the current state of the agent. The variable astate can have one of three values: OK, RE or KO, designating a fault free agent state, a recovery state, and an irrecoverable error correspondingly. We introduce the event AgentRecoveryStart, which is triggered when an agent becomes involved in the error recovery procedure. Observe that AgentRecoveryStart implicitly models two situations:

- when an agent itself detects an error and subsequently initiates its own error recovery,
- when an agent decides to become involved into cooperative recovery from another agent failure.

In both cases the state of the agent is changed from OK to RE.

The event AgentRecovery abstractly models the error recovery procedure. Error recovery might succeed and restore the fault free agent state OK, or continue by leaving an agent in the
recovery state $RE$. Finally, error recovery might fail, as modelled by the event $AgentRecoveryFailure$. The event $AgentRecoveryFailure$ enables the event $AgentFailure$, which removes the irrecoverably failed agent from the corresponding scopes and disengages it from the location.

The introduction of agent states affects most of the events – their guards become strengthened to ensure that only fault free agents can perform normal activities, engage into a location and disengage from it, as well as create and close scopes. In the excerpt from the refinement machine $Cama3$, we present only the newly introduced events and the refined event $AgentFailure$.

**REFINEMENT $Cama3$**
**REFINES $Cama2$**
**SETS** $STATE = \{OK, KO, RE\}$
**DEFINITIONS**

$activeAgent(xx) == (xx \not\in ex_{agents} \land xx \not\in timers)$

**VARIABLES** $\ldots$, $astate$, $recovery_{limit}$

**INVARIANT**

$\ldots \land astate \in agents \rightarrow STATE \land recovery_{limit} \in agents \rightarrow NAT$

**VARIANT** $\sum aa.(aa \in agents \mid recovery_{limit}(aa))$

**EVENTS**

$AgentFailure = \mathrm{ANY} \: aa \: \mathrm{WHERE}$

$aa \in agents \land activeAgent(aa) \land astate(aa) = KO$

$\mathrm{THEN}$

$agents := agents - \{aa\};$ $scopes := scopes \supset \{aa\}$;
$scopes := sowner^{-1} \{\{aa\}\} \subset scopes;$ $sowner := sowner \supset \{aa\}$;
$astate := aa \leftarrow astate;$ $recovery_{limit} := aa \leftarrow recovery_{limit}$

$\mathrm{END;}$

$AgentRecovery = \mathrm{ANY} \: aa \: \mathrm{WHERE}$

$aa \in agents \land activeAgent(aa) \land astate(aa) = RE \land$
$recovery_{limit}(aa) > 0$

$\mathrm{THEN}$

$recovery_{limit}(aa) := recovery_{limit}(aa) - 1$ ||
$\mathrm{ANY} \: vv \: \mathrm{WHERE} \: vv \in \{OK, RE\} \: \mathrm{THEN} \: astate(aa) := vv \: \mathrm{END}$

$\mathrm{END;}$

$AgentRecoveryStart = \mathrm{ANY} \: aa \: \mathrm{WHERE}$

$aa \in agents \land activeAgent(aa) \land astate(aa) = OK \land$
$recovery_{limit}(aa) > 0$

$\mathrm{THEN}$

$recovery_{limit}(aa) := recovery_{limit}(aa) - 1$ ||
$astate(aa) := RE$

$\mathrm{END;}$

$AgentRecoveryFailure = \mathrm{ANY} \: aa \: \mathrm{WHERE}$

$aa \in agents \land activeAgent(aa) \land astate(aa) = RE$

$\mathrm{THEN}$

$astate(aa) := KO$

$\mathrm{END}$

$\mathrm{END}$
As before, in this refinement step we define the system variant to ensure that the newly introduced events converge, i.e., do not take the control forever. To guarantee this, we introduce the variable \( \text{recovery\_limit} \), which limits the amount of error recovery attempts for each agent. Each attempt of error recovery decrements \( \text{recovery\_limit} \). As soon as for some agent \( \text{recovery\_limit} \) becomes zero, error recovery of this agent terminates and the error is treated as irrecoverable. We define the variant as the sum of \( \text{recovery\_limit} \) of agents.

While specifying the error recovery procedure, it is crucial to ensure that error recovery terminates, i.e., does not continue forever. In this refinement step the variant also serves as the means to express this essential property of the system.

Further refinement steps allow us to introduce representation of agent roles into the specification and hence ensure semantic compatibility of agents. Since these steps do not affect the specification of fault tolerance mechanisms, we omit their representation here. Their detailed description can be found elsewhere [LTIR06].

3.5 Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems

3.5.1 Fault Tolerant Replication

System availability is improved by the replication of data objects in a distributed database system. However, during updates, the complexity of keeping replicas identical arises due to failures of sites and race conditions among conflicting transactions. Fault tolerance and reliability are key issues to be addressed in the design and architecture of these systems. In the project we carried out a formal development of a distributed system using Event B that ensures atomic commitment of distributed transactions consisting of communicating transaction components at participating sites [YB06]. This formal approach carries the development of the system from an initial abstract specification of transactional updates on a one copy database to a detailed design containing replicated databases in refinement. Through refinement we verify that the design of the replicated database confirms to the one copy database abstraction.

It is advantageous to replicate data objects when the transaction workload is predominantly read only. However, during updates, the complexity of keeping replicas identical arises due to site failures and conflicting transactions. In addition to providing fault tolerance, one of the important issues to be addressed in the design of replica control protocols is consistency. The One Copy Equivalence criteria requires that a replicated database is in a mutually consistent state only if all copies of data objects logically have the same identical value.

The One Copy Serializability is the highest correctness criterion for replica control protocols [BHG87]. It is achieved by coupling consistency criteria of one copy equivalence and providing serializable execution of transactions [BHG87, OV99]. In order to achieve this correctness criterion, it is required that interleaved execution of transactions on replicas be equivalent to serial execution of those transactions on one copy of a database. The one copy equivalence and serial execution together provide one copy serializability which is supported in a read anywhere write everywhere approach. For example, consider any serial execution of a transaction produced by system in the read anywhere write everywhere replica control. A transaction which writes to a data item does so by writing data everywhere. Thus from the view
point of a transaction which reads the values produced by an earlier transaction, all copies were written simultaneously. So no matter which copy a transaction reads, it reads the same value written by an earlier transaction. Though serializability is the highest correctness criteria, it is too restrictive in practice.

3.5.2 Systems Approach to Development of Replicated Database Systems

In the project we focused on providing a formal analysis of read anywhere write everywhere replica control protocol for a distributed database system. An update transaction which spans several sites issuing a series of read/write operations is executed in isolation at a given site. The basic idea is to allow update transactions to be submitted at any site. This site, called the coordinating site, broadcasts update messages to replicas at participating sites. Upon receipt of update requests, each site starts a sub transaction if it does not conflict with any other active transactions at that site. The coordinating site decides to commit if a transaction commits at all participating sites. The coordinating site decide to abort it if it aborts at any participating site.

We took a systems approach to tackle this problem. Our system model consist of a set of sites and data objects. The distributed database consists of sets of objects stored at different sites. Users interact with the database by starting transactions. The data objects are assumed to be replicated across all sites. The Read Anywhere Write Everywhere replica control mechanism is considered for updating replicas. We consider the case of full replication and assume all data objects are updateable. A transaction is considered as a sequence of read/write operations executed atomically, i.e., a transaction will either commit or abort the effect of all database operations.

The abstract specification maintains a notion of a central or one copy database. It models the entire database as a function from object to values and defines events modelling starting transaction, committing writing or aborting transactions, as well as reading transactions.

In the refinement, the notion of replicated database is introduced via replacing the abstract variable modelling one copy database by the variable modelling replicated database. The corresponding events required to handle replicated database are introduced by superposition refinement. Let us note that one copy equivalence criteria is expressed as a part of gluing invariant in the refinement process. This allows us to formally ensure (by proofs) that conditions defined by the one copy database criterion are satisfied by the operations over the replicated database.

3.6 Discussion

In this section we presented in details only a few approaches which contributed to development of RODIN methodology. These are the works which are the most illustrative examples of an application of systems approach to the development of complex fault tolerant systems. In the project we have collected the solid evidences (reported, e.g., in D 26 – Final report on Case Studies Development and D34 – Assessment report 3) demonstrated the advantages of systems approach. We believe that our work has significantly enriched software engineering field with the methods facilitating system-level thinking.
Chapter 4

Other methodological issues

4.1 Introduction

One important aspect of the RODIN project concerns with the development of an appropriate methodology for formal modelling of systems. This chapter reports on the advancement of methodological aspects during the third year of the project. The developments of the case studies on the Rodin Platform have driven the development of the RODIN methodology. Considering the diverse nature of the different case studies, each of the case studies has contributed to the development of specific aspects of the methodology. Although in the third year, special emphasis was put on validating the Rodin platform and integration of plug-ins, but the methodological issues which were identified earlier have been addressed. In the following sections we summarise the most important methodological aspects which have been addressed during this stage. This document is aimed to summarise the experience of case studies in domain RODIN methodology and offer a strategy for developing similar systems from certain application domain.

4.2 Requirement Modelling and Layering Specification

In modelling real life systems developers have to deal with a huge list of requirements. Devising an appropriate strategy to develop the formal specification from informal requirements is an important stage in the formal modelling process. With each formal specification there are associated a number of proofs which should be discharged. Too much complexity in a formal specification can affect the comprehensibility of the produced model and even with the support of contemporary tools it might take tremendous effort to discharge the associated proof obligations.

During the third year development of CDIS case study on the Rodin platform we partly focused to address the issue that we discussed in the above paragraph. The Initial core specification of CDIS was developed as a single model using VVSL [Mid93] — a variant of VDM [Jon90]. This initial specification has been criticized for the lack of comprehensibility and its susceptibility to mechanised proof. To tackle these problems we had to develop an appropriate approach for formal specification of large systems. Influenced by the step-wise refinement approach recommended as a generic methodology in the new Event-B, we followed a layered approach to gradually integrate all informal requirements into the formal specification.
of CDIS. We call this process “horizontal refinement” to distinguish it from vertical refinement in which a formal specification evolved to a formal design document. In other words, during horizontal refinement, our main aim is to start from a generic abstract specification and produce a complete formal specification. By classifying requirements into manageable subsets and incorporation one subset at a time into the initial specification, we eventually produce a complete formal specification. Design related refinements or vertical refinements can start after the complete formal specification has been produced.

This approach will help to improve the comprehensibility of the formal specification because we can choose an appropriate level of abstraction to look at. In the meantime, the layered structure of the specification means that in each level we have to deal with a limited number of proof obligations. As we mentioned this approach has been adopted during development of CDIS case study as an example of industrial system and well-documented in Section 5 of deliverable D26.

4.3 Complex Data Types and Layered Refinement

During formal modelling we may need to model structured data types. One important aspect of structured data type modelling is that –in most cases– we do not need to introduce all of the details of the structured data in one step: this is to avoid unnecessary complexity and keep the model as simple as possible. In compliance with the general approach of layered refinement in Event-B, a layered approach to modelling of structured data was devised. Also this approach has been applied in the context of the CDIS case study but it is general enough to be applied to all similar cases.

An important aspect of CDIS redevelopment with the Rodin tools is the modelling of structured data. The undertaken approach which has been reported in [EB06] is based on deferred set and constant functions. In this approach the structured data would be defined as an abstract set. This serves as the type definition of the structured date or record. According to the modelling needs, the record will be refined by introducing the individual fields. The introduction of necessary fields will take place in a stepwise manner by the means of constant functions. The main advantage of this method is that we introduce new record’s fields whenever it is necessary. In addition this approach is compatible with The B-Method general refinement approach. In the rest of this section we discuss the two different paths that we have taken to refine the record types in the Rodin tool.

Both in the VDM and second year B models, we have structured data types in the form of records. In the second year B models instead of introducing the whole structure at once we have gradually introduced different fields when they were needed. The main motivation of this approach is to enable a stepwise development of complex record structures (in the spirit of refinement) by introducing additional fields as and when they become necessary.

One possible style of using abstract types as records has been followed in the second year models. This approach is based on delaying the introduction of a record type to later stages of refinement. In some cases at the abstract level, we might be unaware that a simple (non-record) state variable requires a record structure at a later stage in the development. Hence in the most abstract level we have a set of simple abstract types which they have been defined as a deferred set. For example to define the central database of the CDIS system at a very abstract level we have defined it as a total function from \(Attr\_id\) to \(Attr\_value\). Both of \(Attr\_id\) and \(Attr\_value\) are
sets. In the refined model we have replaced the `Attr_value` with a new type named `Airport_attr`. Now the new type is a record as following:

```
Airport_attr ::  value : Attr_value
                Last_update : Date_time
```

After this refinement the next step is to amend the type of any variables or local parameters which have been affected. For example in the case local parameters which have been defined in ANY statements, we have to define the relation between the abstract parameters and the refined one through the use of witness clauses. Also we found that in many situations the tool can handle the related proof obligation quite easily but in some cases discharging related proof obligation is not so straightforward. Therefore in the third models we have followed a different approach to model records.

In this adopted approach, during refinement stage, we do not change the name of record type. Instead of this we introduce the necessary fields through the use of constant mappings. For example we define the database of the previous case as:

```
Database = Attr_id → Attrs
```

And then we define the constant function to relate a value to an attribute.

```
value ∈ Attrs → Attr_value
```

The main advantage of this approach is that we do not need to use a witness to define the relation between the refined and abstract parameters. In addition to this, it will eliminate the need for having extra invariants. These make the generated proof obligations simpler and as a result some interactive proofs now could be discharged automatically without any user interactions. Another advantage is that increased the comprehensibility of our formal models. This approach has been reported in D26.

## 4.4 Proofs and Gluing Invariants Discovery

In this section we outline the way in which the gluing invariants can be discovered by investigating the generated proof obligation. In many situations when we construct a formal refinement, linking the refinement to its previous level model is not always a straightforward task. One very useful approach that has worked in many cases is to rely on the interactive prover to discover the gluing invariants. The basic rule of this method is quite simple and we will discuss it very briefly in the following section.

Initially construct your refinement with just a basic typing invariant and no gluing invariant. The first pass of automatic proof can resulted in several proof obligations that could not be proved automatically. In the next stage the developer needs to examine the generated proof obligation carefully. The `goal` part of in the unproved proofs should give the developer an overview of what form of gluing invariants is needed. When the first round of invariants are added to the model, new proof obligations associated with these new invariants are generated. In order to discharge these additional proof obligations we may need to add another set of
invariants to our model. After few iterations of invariant strengthening, we should arrive at
set of invariants that are sufficient to discharge all the proof obligations automatically. An
appropriate gluing invariant is the key to prove the correctness of a refinement step. In this
section we outlined how we used the interactive prover to guide us in constructing a gluing
invariant. As well as easing the burden of inventing the gluing invariant, this approach also
has the consequence that the form of the gluing invariant we use closely matches the form of
the proof obligations thereby making the mechanical proofs much easier and in many cases
completely mechanised. For a more detailed overview of this approach interested readers can
refer to [BY07].

4.5 Adjusting Modelling Style to Achieve Higher Productivity

In this section we summarised some cases involving minor tuning in the modelling styles which
can have noticeable effects on the generated proofs. Again these issues have been applied in
the context of CDIS case study but one can expect to adapt it for any similar cases.

In the context of CDIS modelling on the Rodin platform some adjustments have been made
into models to achieve higher level of automatic proof or increase the readability and com-
prehensibility of the B models. Also these changes might be seen trivial — but based on our
experiment with the RODIN tool they had a great effect either on the comprehensibility of our
models or level of automatic proof discharging or even both of them. Some of these changes
in style of modelling may be not directly tailored to the RODIN tool, but from our viewpoint
it is very important to document them. This can help other developers to take the advantage of
these subtle techniques to improve their modelling.

The first style change that we want to point out here is using relation instead of power set.
In refinement level 3 of the Year 2 models we have the following declaration:

\[
edd_{acks}\ required \in EDD_id \rightarrow P(Attr_id)
\]

that we have changed to:

\[
edd_{acks}\ required \in EDD_id \leftrightarrow Attr_id
\]

These two declarations are almost identical from the mathematical viewpoint but from a
modelling viewpoint the story is different. In a very simple comparison between the third and
second year models it can be seen that this change has resulted in significant simplification in
the events which manipulate this variable. For example complex lambda notation and nested
restriction has been reduced to simple composition. The first effect of this is on the comprehen-
sibility of the model which has been increased. Secondly, it has simplified the generated proof
obligation in such a way that either they could be discharged automatically or with minimum
intervention from the user which was not the case with previous style.

The second aspect of style related issue is using clarification declaration. In many situations
especially when we use local parameters in ANY statements the type of parameter could be
implicitly defined through the guards. If the guard if fairly simple it is very easy to find out the
type of local parameter. In many practical situations this is not the case and comprehensibility
of the model will increase if you add a clarification declaration for these parameters. In our
RODIN-based models we have used the method to assist the potential viewers of our model. In
addition to this it has helped us to deal with the interactive proofs more easily.

4.6 Augmenting Semiformal Notation With Formal Notation

Using UML as a graphical based notion is widespread and well accepted in the software indus-
try. One major weakness of such approaches is the lack any underlying formal semantics
that could help to verification task of produced models. One major direction in recent years’
research was to combine UML with other formal notations to produce a set of formally verifi-
able models either directly or indirectly. In the context of the RODIN project, this constituted
a significant part of the project in the form of developing related plug-ins.

These efforts comprised different activities including developing a combined notation known
as UML-B and related methodology and supporting tools. The core part of tool support to
translate UML-B models into Event-B models is provided by U2B. Using these facilities in the
Rodin platform provide a significant method to formalise and use UML in a more structured
and rigorous way. UML as a whole suffers from an incomplete, inconsistent semantics which
makes accurate application of the UML problematic at best. Bridging the gap between lan-
guages such as UML and Event-B paves the way for wider acceptance of Event-B and Rodin
platform in industrial environments. U2B overall provides a good compromise between the
mathematical abstractness of Event-B to the apparent lack of verifiability in UML. More de-
tails on how this approach has been used in the context on RODIN case studies can be found in
D26.

4.7 Verification Versus Proof-based Modelling

Animation and verification of the initial specification before attempting the theorem-based
prover in many situations has proved to be an effective approach. This facilitates finding
any flaws and gaps in the specification before wasting a lot of time on struggling with un-
dischARGEable proofs. This facility has been provided by the ProB Plug-in on the Rodin plat-
form.

ProB provides much necessary support for animation and verification of Event-B models. Using ProB was reported very useful in validating early specification models. Verification
removes certain error and ensures that the proof associated with model would be discharged
more smoothly by the Rodin prover. Animating the model in the ProB enables us to establish
whether or not the formal specification met the customer’s demands and comply with system’s
requirements. This fits in well with the style of development commonly seen in industry where
constructing a model to investigate the properties of the system and their feasibility. ProB and
the its supported validation style of development in this sense provides a way of first construct-
ing and demonstrating systems then discovering properties later. The Animation of the model
at this can be used in order to investigate failed POs. The ProB disprover can be used to pro-
duce a counter example to a proof obligation, which may help strengthening any related guards
or invariants.
In addition to this the use of ProB is particularly useful with regards to the people who want to start exploring the new Event-B language and Rodin Platform. Practitioners may find it more natural and simpler to animate and verify their model first and when they gained more confident they can start using the proof-based approach later. Again the D26 document can be a useful source of further information regarding the application of ProB in different case studies.

4.8 Conclusions

In this chapter we have reviewed a range of different methods and approaches based on the new Event-B notation and its supporting tool. These methodological aspects are believed to be general enough that they can be applied to similar cases. We hope this will provide guidelines for whoever wants to gain a basic understanding of the field and start modelling with Event-B and use the platform.
Chapter 5

Specifics from the case studies

This chapter sets out how the general methods described in the previous chapters have been merged with those of the groups developing the case studies. We repeat that we now see this as an essential ingredient of a DEPLOYment strategy.

5.1 Case Study 1: Formal Approaches in Protocol Engineering

The work in the third year of RODIN on the case study on “Protocol Engineering” is described in Section 2 of D26 along with an overview of its achievements during the whole project. Of particular interest is the methodological link of the Lyra research [LTL+06c] to refinement in B [LTL+05] and Deliverable D8; fault-tolerance is discussed in [LTL+06c].

Specific progress in year three of RODIN includes: concurrency issues see Laibinis et al. in [BJRT07]; use of the RODIN platform; and a formal basis for the Lyra B models see Malik et al. in [BJRT07]. Further details (in the context of the case study) are given in Section 2.2 of D26. Of particular interest are the developing notion of consistency of Lyra models and modelling recovery (after fault) actions.

5.2 Case Study 2: Engine Failure Management System

Section 3 of D26 describes the work on the “Engine Failure Management System” (FMS). As its name suggests, this case study is ideal for fault-tolerant (“resilience”) considerations; the additional “Production Acceptance Test” (PAT) case undertaken in year three is also described in Section 3 of D26.

As described in Section 3.1.2 of D26, most of the work on FMS was actually complete by year two of Rodin and is reported on in D9 and D19. Noteworthy are the successful use of a “Traceable Requirement Document”; the deployment of the ProB model checker; and the “instantiation validation” (this led in year three to a support plug-in for “Context Management”).

The PAT work is described in Section 3.1.3 of D26. The main methodological issues here were on genericity and dynamic failure management.

It is important to remember that staff at the company (ATEC) who undertook this case study had limited FM experience. As was our intention when we set out the DoW, this gives 39
us a different sort of evaluation (see D34) of the RODIN methodology. Without trespassing
on the territory of D34, it is reasonable to claim a success in this deployment of the Rodin
methodology and tools.

5.3 Case Study 3: MDA

The evolution of Nokia’s business needs has been reported in D33 along with the impact that
these changes had on the MDA case study. Basically, Nokia concluded the NoTA project at the
end of 2006. It is of course common for industrial entities to change their directions within the
timescale of such a project and it is to Nokia’s credit (and that of the Project Coordinator) that
RODIN got as much out of this case study as it did.

The results are set out in Section 4 of D26: specific topics (Section 4.2.1 of D26) include:
fault-tolerance (use of MDA), UML/B, model-based testing, evolution of requirements,

5.4 Case Study 4: CDIS

The “CDIS” case study is described in Section 5 of D26. Much of the activity in year three
has been concerned with the use of the RODIN tools (to replace the earlier use of “B4free”) but
there has also been considerable effort on looking at the way the year two models facilitate
reusability, traceability and adaptability.

The impact of the move from “pure” B to Event-B is discussed in Section 5.2.1.2 as is the
approach to “records”.

This work considered both an idealised (in the sense of assuming a single database) and a
distributed version. Atomicity concerns were addressed by “event splitting”.

A key outcome and success of this work is the use of a generic model — separating “con-
text” and dynamic parts. The figures on the automatic discharge of POs are also very en-
couraging — cf. Section 5.2.1.4. We are –of course– fully aware that that such high levels of
automatic support come about only by careful layering and it is unlikely that less expert users
will immediately achieve such good results.

As was set out in the initial DoW, a key reason for undertaking this study was the previous
work on CDIS (by Praxis) using a variant of VDM [Jon90] known as VVSL [Mid93]. Although
the model presented in Event-B is both much clearer and more tractable, the Rodin project
members are not claiming that the entire difference is down to the Rodin language (Event-B)
and methods. It is always possible to improve on a formal model; in this case the improvement
has been very dramatic by extremely clever factoring of ideas.

5.5 Case Study 5: Ambient Campus

This case study pushes the RODIN methodology in an attempt to cover “ambiance” (Mo-
bile Agent Systems); here, the main faults to be tolerated are concerned with transmission
errors/failures. The progress is described in Section 6 of D26 where several “scenarios” are
discussed.
A particular focus of the third year has been on the use of “patterns” to reduce the amount of formal proof (proof obligations) required for each new application (and, indeed, the number of mistakes made during design).

Section 6.2 of D26 sets out the main conclusions from the Ambient Campus study. The work in [IKKR05] and Iliasov et al. in [BJRT07] presents a combination of Event-B with aspects of process algebraic notation. The use of—and impact on—RODIN platform and plugins is also described. Interestingly, this also discusses testing.
Chapter 6

Other items of report

6.1 Key meetings

Meetings associated with RODIN methodology were:

- A project internal event was held at Nokia Research in Helsinki in January 2007: this was extremely fruitful and led to the decision to report the methodological aspects of the Case Studies in their respective sections of D26.

- An open workshop on “Methods, Models and Tools for Fault Tolerance” was held in Oxford in conjunction with the “Integrated Formal Methods” symposium in July 2007. The proceedings are available as [BJRT07]. We have agreement from Springer Verlag again (see next item) to publish a post-conference set of papers that are developed from the workshop presentations and includes invited contributions from leaders in the field of formal methods for fault tolerance.

- The proceedings of the earlier conference are available as [BJRT05]; the corresponding collection of developed and invited papers has now been published as [BJRT06].

- Elena Troubitsyna presented a very clear overview of the RODIN methodology work at the second RODIN Industrial Awareness day (Paris, September 2007).

6.2 Publications

New publications/reports relating to methodology: [Pop07, SPJ08, JHJ07, HH07, Meh07, RK07, JLW07, KK07, KKKV06, DKK06c, KPPK07, Jon07b, JP07, CJ07] (the bibliography below includes earlier publications). The proceedings of the VSTTE conference held at ETH in October 2005 contain papers from several of those involved in the project. Unfortunately, the proceedings have been unaccountably delayed for an inordinate length of time (no one in the RODIN project was responsible for any of this delay). The proceedings are finally “in press” as Number 4171 of Springer’s LNCS series.
Chapter 7

The way forward

Many of the technical challenges that we see for the future of applying formal methods as a help for designing and developing fault-tolerant (or resilient) systems are described in the Project Proposal for the DEPLOY (FP-7) project.

A good example of an issue on which the industrial partners in that project will need support is handling probabilistic requirements: it is almost inevitable that fault-tolerance has to be discussed in terms of things like MTBF. Although there is a significant corpus of research papers on probabilistic notations, we feel that there is basic thinking required before it fits smoothly into something close to the RODIN methods.

Another area that we expect to pursue concerns scaling of both methods and tools. We have recognised with the interactions within RODIN that a major issue for the adoption of both methods and support tools is how readily they can be integrated with procedures being used within the organisations.

Lastly, we have been absolutely clear in discussions with the DEPLOY collaborators that any generic methods will have to be merged with the current methods and tools of the particular organisation where DEPLOYment occurs.
Bibliography


[LIM+05] Sari Leppänen, Dubravka Ilic, Qaisar Malik, Tarja Systä, and Elena Troubitsyna. Specifying UML Profile for Distributed Communicating Systems and Communication Protocols. Proceedings of Workshop on Consistency in Model Driven Engineering (C@MODE’05), November 2005.


