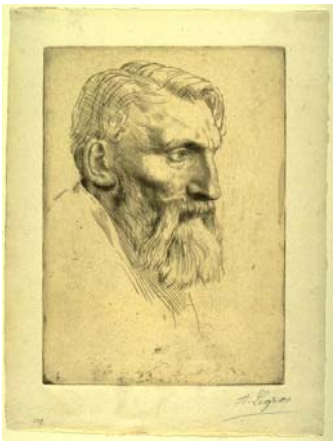
Project IST-511599
RODIN
"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable D18

Intermediate Report on Case Study Development

Editor: *Elena Troubitsyna (Aabo Akademi University, Finland)*

Public Document

31st August 2006

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

Contributors:

Budi Arief (University of Newcastle upon Tyne, UK),
Pontus Boström (Aabo Akademi University, Finland),
Alex Iliasov (University of Newcastle upon Tyne, UK),
Dubravka Ilic (Aabo Akademi University, Finland),
Ian Johnson (AT Engine Controls Ltd, UK),
Maciej Koutny(University of Newcastle upon Tyne, UK)
Linas Laibinis (Aabo Akademi University, Finland),
Sari Leppänen (Nokia, Finland),
Mats Neovius (Aabo Akademi University, Finland),
Ian Oliver (Nokia, Finland),
Mike Poppleton (University of Southampton, UK),
Alexander Romanovsky (University of Newcastle upon Tyne, UK),
Mannu Satpathy (Aabo Akademi University, Finland),
Colin Snook (University of Southampton, UK),
Elena Troubitsyna (Aabo Akademi University, Finland),
Marina Waldén (Aabo Akademi University, Finland)

Contents

SECTION 1. INTRODUCTION

This document reports on the second year of the development of case studies in RODIN. The case studies drive the development of the RODIN methodology and supporting platform, validate it and evaluate its cost-effectiveness. In this deliverable we describe the results achieved over the last year and outline the future plans.

In general, the development of the case studies proceeds according to the original plan. Each of the case studies is contributing to the development of both the methodology and the supporting platform. A number of methodological issues identified in year one have been addressed in the second year. We have also started the work on validating the tool platform. The case studies have been actively driving development of the tool plug-ins. Besides contributing to the development and validation of initially planned plug-ins, several new plug-ins have been proposed as a response to challenges discovered in the case study developments.

In Section 2 we describe the advances made in the development of case study 1 – *Formal Approaches to Protocol Engineering*. The case study investigates the use of formal methods in model-driven development of communicating systems and communication protocols. Over the last year this work was proceeding in two major directions – formalization and extension of the existing UML2-based development method Lyra and formal verification of the Lyra metamodel. We describe the progress in augmenting Lyra with formal reasoning about fault tolerance advances in integrating model-based testing as well as verification of the Lyra metamodel by refinement in B.

Section 3 presents the progress achieved in case study 2 – *Engine Failure Management System*. The aim of the case study is to study how the methods and tools developed in RODIN could improve design, maintenance and re-use of the failure management systems developed by ATEC[1]. Observing the results of year one, the reviewers proposed to leverage industrial evaluation of RODIN technology and increase level of the ATEC expertise in formal modelling. To respond to this observation, during the second year, ATEC performed a pilot evaluation study. We briefly present the experience gained from it. Moreover, we report on continuing the work on development, instantiation and reuse of the generic model developed in year one. To support this line of research, a prototype plug-in, 'the Requirement Manager', was designed and implemented. Furthermore, in this deliverable we also present the result of developing failure management systems by classical refinement in B. The development has been undertaken by an academic partner with the aim to transfer knowledge of formal modelling and design to the industrial partner.

[1] AT Engine Controls Ltd, U.K.

In Section 4 we describe the developments in case study 3 – *Formal Techniques within MDA Context.* In the second year we have further investigated how the RODIN techniques and tools can be applied in a model based environment and work flow by using them to develop of a hardware based mobile phone platform known as NoTA. In this deliverable we describe the advances made in combining modelling techniques of UML, B, and hardware description languages to specify HIN – High Interconnect layer of NoTA. Moreover, the use of animation, CSP and model-based testing, and theorem proving for verifying the HIN layer is investigated.

In Section 5 we give an overview of work on case study 4 – *CDIS Air Traffic Control Display System.* In the second year we have focused on developing a methodology for constructing large formal specifications, eligible for tool-based formal analysis. The major problem spotted in the CDIS development a decade ago was a lack of continuity from the specification to design. Namely, the idealized view taken in the core specification – an instantaneous update of information in all nodes – could not be mapped into the implementation, which had to take into account inevitable communication delays in information distribution. In this deliverable we report on patterns and generic techniques for modelling and refinement developed to alleviate this problem.

Finally, in section 6 we reflect on the experience gained during the second year of work on case study 5 – *Ambient Campus*. The aim of this case study is to investigate the use of formal methods combined with advanced fault tolerance techniques in developing highly dependable ambient intelligence applications. One of the directions of the work is development of fundamental abstractions for formal development and verification of ambient systems. We briefly describe the issues in formal development of distributed middleware and present the set of formal decomposition patterns assisting development of multi-agent ambient systems. Moreover, we outline our work on exception handling in this domain and advances in the development of a plug-in for model-checking ambient systems.

In general, during the second year the case studies have continued to actively drive the development of the methodology and supporting tool platform. The joint research efforts initiated in year one have been further expanded and strengthen over the last year. We believe, that this provides us with a strong basis for successful accomplishment of the goals set for the final year of the project.

SECTION 2. CASE STUDY 1: FORMAL APPROACHES IN PROTOCOL ENGINEERING

2.1 Introduction

This section summarises the developments of Case study 1 – "Formal Approaches in Protocol Engineering" – during the second year of the RODIN project. The goal of CS1 is to investigate the application of formal methods for development of telecommunication systems and communicating protocols [2.11]. The work on the case study focuses on formalisation and verification of the design method Lyra developed in the Nokia research center. Lyra is an UML2-based service-oriented method for development of telecommunication systems and communicating protocols. Within RODIN we aim at providing support (in the form of formal techniques and tools) for various stages of this approach.

During the first year of the RODIN project we have developed formal specification and refinement patterns reflecting essential Lyra models and transformations. This allowed us to conduct verification of the Lyra development using stepwise refinement in the B Method. This work has been reported in [2.6, 2.10].

In the second year of the RODIN project our work on the case study has progressed in three directions:

1. Incorporating formal reasoning about fault tolerance into the formalized Lyra development flow [2.5];
2. Developing an approach to verifying the consistency of the provided UML models [2.7];
3. Developing preliminary methodology for model-based testing of Lyra B models.

The work on CS1 in year 2 has been presented in a series of internal RODIN workshops and presentations listed below.

- Presentation at Zurich plenary meeting (September 2005);
- Presentation to EU commission (October 2005);
- Presentation at Aix-en-Provence workshop (April 2006).

In addition, the achieved results were presented at the following international conferences and workshops:

- International Conference on Formal Engineering Methods (ICFEM'05) – the work on verification of Lyra by B refinement [2.6],
- Workshop on Consistency in Model Driven Engineering (C@MODE'05) – the work on verifying the consistency of the provided UML models [2.7].

2.2 Major Directions in Case Study Development

Next we describe the contribution of the case study to methodology and plug-in development achieved in the second year of the RODIN project. During this year our work has focused on the task **T1.1.4**:

T1.1.4 Investigate the use of refinement and model checking to verify decomposition and composition steps. Investigate the combination of model checking and refinement techniques in context of UML and B. Investigate the use of model checking tools in combination with UML to B tool. Investigate the applicability of formal reasoning about fault tolerance in this application area.

defined in [2.11]. We have progressed in three major directions described below.

First, to incorporate formal reasoning about fault tolerance into the formalized Lyra development flow, the specification and refinement patterns for Lyra models have been extended with explicit representation of possible errors and error recovery. The extension has affected the specifications of service components directly responsible for controlling the service execution flow (called service directors). The recovery mechanisms allowing a service director to retry the failed service execution as well as to "roll back" in the service execution flow have been incorporated in the specification of a service director. Moreover, in the refinement steps modelling service decomposition and distribution over a given network, the fault tolerance mechanisms have been distributed over the involved service components. Termination of potentially infinite recovery process has been guaranteed by modelling the maximal execution time that is gradually decreased by service execution.

Second, to automate translation and verification of Lyra UML models in the B Method, we have developed an approach to verifying the consistency of the provided UML models. The approach consists of formalisation of the intra-consistency (i.e., expressing the relationships between models within the same Lyra development phase) and inter-consistency (i.e., the relationships between different Lyra phases) rules for the Lyra UML models. The formalisation is done using the B Method in such a way that the requirements are gradually (i.e., phase by phase) introduced and incorporated by

the corresponding B refinement steps. The achieved results create a basis for developing a formally verified UML profile for Lyra.

Third, to investigate the use of model checking and model-based testing techniques in the context of UML and B, a preliminary methodology for model-based testing of Lyra models has been developed. Once Lyra UML models are translated into the corresponding B models, a finite coverage graph can be created by the model checking tool ProB. Some paths starting from the initial state of this graph are then taken as test cases. Finally, an extension of ProB called ProTest can simultaneously run both the B model and the corresponding Java implementation using the test cases generated in the previous step.

We now present these results in more detail.

2.2.1 Introducing Fault Tolerance in the Lyra Development Flow

Initially the Lyra methodology has addressed fault tolerance implicitly, i.e., by representating failed service provision in the system models without modelling the mechanisms for fault detection and recovery – the fault tolerance mechanisms. We argue that by integrating explicit representation of the means for fault tolerance into the entire development process, we establish a basis for constructing systems that are better resistant to errors, i.e., achieve better system dependability. Next we will discuss how to extend Lyra to integrate modelling of fault tolerance.

Lyra consists of four main phases: Service Specification, Service Decomposition, Service Distribution and Service Implementation. The *Service Specification* phase focuses on defining services provided by the system and their users. In the *Service Decomposition* phase the abstract model produced at the previous stage is decomposed in a stepwise and top-down fashion into a set of service components and logical interfaces between them. In the *Service Distribution* phase, the logical architecture of services is distributed over a given platform architecture. Finally, in the *Service Implementation* phase, the structural elements are integrated into the target environment and platform-specific code is generated.

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

In the *Service Decomposition* phase we decompose the service provided by a service component into a number of stages (subservices). The service component can execute certain subservices itself as well as request the external service components to do it.

According to Lyra, the flow of the service execution is managed by a *Service Director* (often called a Mediator). It implements the behaviour of the service component as well as co-ordinates the execution flow by enquiring the required subservices from the external service components.

In general, execution of any stage of 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 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

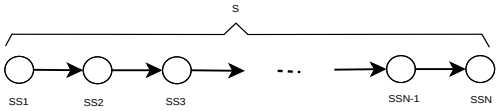
- retry the execution of the failed subservice,
- repeat the execution of several previous subservices and then retry the failed subservice,
- abort the execution of the entire service.

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 errors lead to aborting the entire service. In **Fig.2.1(a)** we illustrate a fault free execution of the service *S* composed of subservices $S_1, \ldots, S_N$. The execution of *S* in presence of various kinds of errors is shown in **Fig.2.1(b) - 1(d)**.

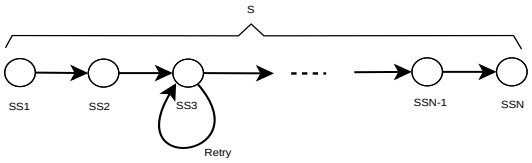
Let us observe that each service should be provided within a certain finite period of time – the *maximal service response time Max_SRT*. In our model this time is passed as a parameter of the service request. Since each attempt of subservice execution takes some time, the service execution might be aborted even if only recoverable errors have occurred but the overall service execution time has already exceeded *Max_SRT*. Therefore, by introducing *Max_SRT* in our model, we also guarantee termination of error recovery, i.e., disallow infinite retries and rollbacks, as shown in **Fig.1(e)**.

To derive the specification of *Service Director* for the *Service Distribution* phase, we have to take into account a given network architecture. In general, we should consider two cases:

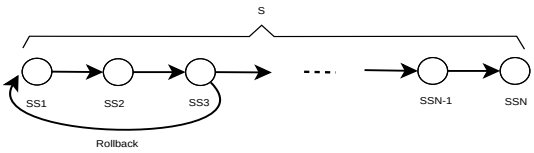
1. *Service Director* is "centralized", i.e., it resides on a single network element,
2. *Service Director* is "distributed", i.e., different parts of the execution flow are orchestrated by distinct service directors residing on different network elements.



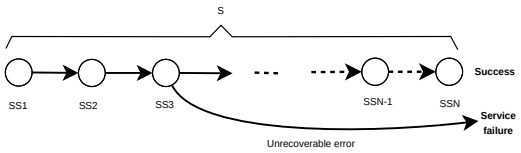
(a) Fault free execution flow



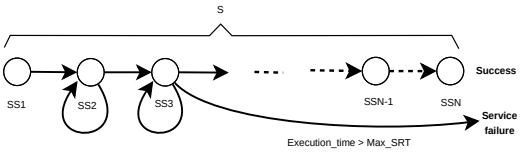
(b) Error recovery by retrying execution of failed subservice



(c) Error recovery by rollbacks



(d) Aborting the service execution



(e) Abort of the service due to timeout

Fig. 2.1: **Service decomposition: faults in execution flow**

Assume for simplicity that the set of subservices required to provide *S* consists of three elements: S_1, S_2 and S_3. At the *Service Decomposition* phase, in both cases the model of the service component providing the service *S* looks as shown in **Fig.2.2**. The service distribution architecture diagram for the first case is given in **Fig.2.3**. In the second case, let us assume that the execution flow of the service component is orchestrated by two service directors: the *Service Director1*, which handles the communication with the external users and also communicates with the service component providing S_1, and *Service Director2*, which orchestrates the execution of the subservices S_2 and S_3. The service directors communicate with each other while passing the control over the corresponding parts of the flow. The architecture diagram depicting the overall arrangement for the second case is shown in **Fig.2.4**.

In the Lyra B development, decomposition and distribution of services are modelled as separate refinement steps. In the Service Decomposition phase we introduce the abstract function *Next* that describes the Lyra execution flow from the point of view of a service director. This function is used by a service director to decide which service is

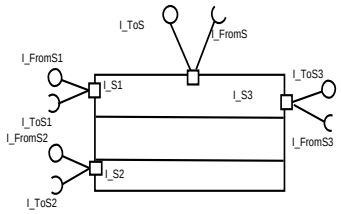


Fig. 2.2: **Service component**

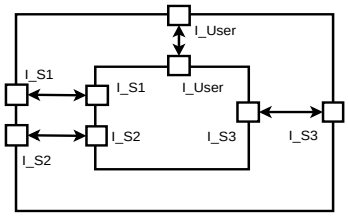


Fig. 2.3: **Architecture diagram (case 1)**

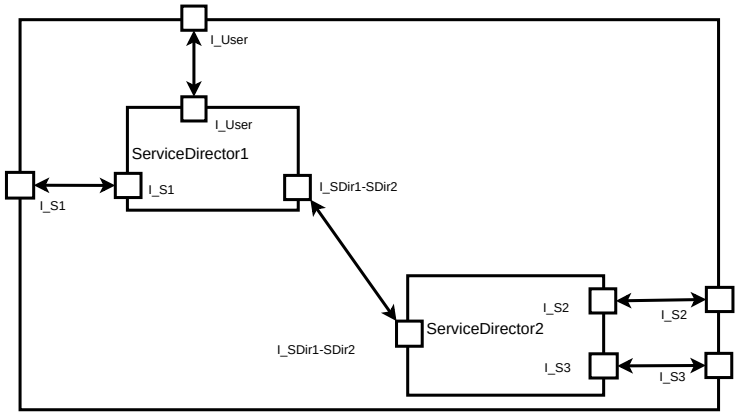


Fig. 2.4: **Architecture diagram (case 2)**

to be executed next in the absence or the presence of faults. In the Service Distribution phase the function *Next* is instantiated taking into account a given network architecture. The detailed B specification patterns of service components with incorporated fault tolerance mechanisms are presented in Section 2.3 on demonstrators.

In the year 2 of the RODIN project we have extended the Lyra B specification and refinement patterns with explicit representation of possible errors and error recovery. Moreover, in the refinement steps modelling service decomposition and distribution, the fault tolerance mechanisms have been distributed over the involved service components. In the year 3 we are going to further enhance the Lyra B models by modelling parallel execution and dynamic reconfiguration of services.

2.2.2 Verifying Consistency of Lyra UML models

One of the goals of CS1 is to develop a tool for automatic translation of Lyra UML models into the corresponding B specifications. To achieve this, the consistency of provided UML models should be formally verified. We start formal verification of consistency by deriving the list of informal requirements for Lyra UML models. In particular, for each Lyra stage we derive the list of requirements corresponding to a

particular Lyra model. For each model we group requirements around concrete model elements. Once the complete list of requirements is obtained, we can distinguish between model-presentation, intra-, and inter-consistency rules for each particular Lyra model.

The informal requirements form the basis for formalizing Lyra models and consistency rules in B. In general, the approach is as follows. For each Lyra model we introduce the corresponding B machine specifying the way the model is constructed. The B machines are created in the order defined by the Lyra development flow. Hence, the set of models defined at each stage is represented by the corresponding set of B machines. The intra-consistency rules are defined as the invariant of a top machine – a machine which includes this set of B machines. The models at each subsequent stage are represented in the same way. Moreover, inter-consistency is ensured by refinement between the corresponding top machines. The refinement relation defined as a part of the invariant of the top machine contains inter-consistency rules. Next we present our approach in detail.

Ensuring intra-consistency of Lyra models in B. Ensuring intra-consistency in Lyra requires verifying that the models:

- satisfy *model presentation rules*, i.e., constraints expressing how to properly define its elements, and
- are *not contradictory* with each other.

To achieve verification of these properties, we first represent each kind of Lyra models as a B machine of a general form given in **Fig.2.5**. The name of the machine corresponds to the name of the Lyra model and is followed by the acronymic name of the stage, i.e., SS, SDe or SDi. The variables of this machine correspond to model elements and their presentation rules are expressed as its invariant.

The machine operations simulate creation of model elements. Namely, for each model element there is one corresponding **Create_ModelElement** operation which allows the creation of the element by enforcing at the same time the model presentation and the intra-consistency rules.

To ensure that the models are created in a certain order we introduce the variable *Model_Stage_Status*. While creating the corresponding Lyra model, the operation **Start_Model_Stage** assigns the value *Creating* to *Model_Stage_Status* and this, in turn, enables the creation of elements of the model. Let us observe that *Model_Stage_Status=Creating* is the guard of the **Create_ModelElementA** and **Create_ModelElementB** operations in **Fig.2.6**. When a particular model is created, *Model_Stage_Status* variable is assigned value *Finished*.

MACHINE *Model_Stage*
EXTENDS $<$ *Previously created model* $>$

VARIABLES $<$ *Names of model elements* $>$, *Model_Stage_Status*
INVARIANT $<$ *Model presentation rules* $>$
INITIALISATION
$<$ *Initialise the variables for model elements* $>$ $\|$ *Model_Stage_Status* := *Empty*
OPERATIONS
Start_Model_Stage =
BEGIN
Model_Stage_Status := *Creating*
END;
Stop_Model_Stage =
SELECT $<$ *Model creation rules satisfied* $>$
THEN
Model_Stage_Status := *Finished*
END;
Create_ModelElementA =
SELECT *Model_Stage_Status* = *Creating*
THEN
$<$ *Create a model element A while ensuring model presentation and intra − consistency rules* $>$
END;
Create_ModelElementB =
SELECT *Model_Stage_Status* = *Creating*
THEN
$<$ *Create a model element B while ensuring model presentation and intra − consistency rules* $>$
END;
...
END

Fig.2.5: General form of the B machine for Lyra model

The creation of models at each particular stage is orchestrated by the corresponding top machine. Its general form is shown in **Fig.2.6**. After one model is created, the top machine corresponding to that stage defines which model is to be created next. Namely, if *Model2* should be created after *Model1* at the stage I then the guard of the **Create_Model2_Stage** operation of this machine has the following form:

$$Model1\_StageI\_Status = Finished \ \wedge \ Model2\_StageI\_Status = Empty$$

where the value *Empty* assigned to the variable *Model2_StageI_Status* indicates that the creation of *Model2* has not started yet. The creation of *Model2* is then triggered by the operation **Start_Model2_StageI** called from the body of the operation **Create_Model2_StageI**.

Since we assume that the Lyra models are checked for consistency only after they are created, the invariant of the machine corresponding to a certain Lyra stage guarantees that the intra-consistency rules for a particular model are satisfied only when *Model_Stage_Status=Finished*.

To verify the intra-consistency rules, we should prove correctness of the defined top machines and abstract machines representing Lyra models.

Ensuring inter-consistency of Lyra models in B. To verify inter-consistency, we should ensure that the models at different development stages are not contradictory with each other. In this section we propose refinement as a technique for establishing model inter-consistency.

The models from each Lyra stage correspond to the B machines specified according to the pattern given in **Fig.2.5**. The rules of intra-consistency remain unchanged through stages. However, the models starting from the second Lyra stage are obtained based on the models from the previous stage. A B machine corresponding to the top machine of subsequent Lyra stage is a refinement of the top machine for the previous Lyra stage and its general form is shown in **Fig.2.7**.

The top machine *StageII* uses a specific form of data refinement called superposition refinement [2.1]. Superposition refinement introduces new variables while leaving the existing data structure unaffected. Observe that the general ideas of superposition refinement and model transformation during the Lyra development process coincide. Each development stage introduces a new set of models, while the models created at the previous stage remain unchanged. The way that elements of the models from one stage relate to the elements from the models in another stage defines the inter-consistency rules between these two stages. These rules are enforced while creating the elements of Lyra models in the subsequent Lyra stages.

Although the refinement *StageII* has the form similar to that of the machine *StageI* (see **Fig.2.6**), the invariant of the refinement *StageII* additionally expresses not only the intra- but also the inter-consistency rules. The inter-consistency rules are expressed as the linking invariant of the refinement *StageII*. To verify the inter-consistency rules, we should prove correctness of defined abstract machines corresponding to the models of the subsequent stage and the refinement of this stage.

The proposed methodology is developed using the Classical B and verified with its automatic tool support – AtelierB [2.2]. The newly developed Event B does not suppoort the machine inclusion mechanism extensively used in our approach. However, the proposed development is based on the superposition refinement (i.e., adding new data structures and operations), which can be easily modelled in the Event B.

MACHINE *StageI*
EXTENDS *Model1_StageI*
INVARIANT
$/* \ intra - consistency\ rules \ */$
$/* \ Model1 \ */$
$(Model1\_StageI\_Status = Finished \Rightarrow ...)$
$/* \ Model2 \ */$
$(Model2\_StageI\_Status = Finished \Rightarrow ...)$
OPERATIONS

Create_Model1_StageI =
SELECT
Model1_StageI_Status = *Empty*
THEN
Start_Model1_Stage
END;
Create_Model2_StageI =
SELECT
Model1_StageI_Status = *Finished* $\wedge$
Model2_StageI_Status = *Empty*
THEN
Start_Model2_Stage
END
...
END

Fig.2.6: General form of a B machine for a specific Lyra stage

REFINEMENT *StageII*
REFINES *StageI*
EXTENDS *Model2_StageII*
INVARIANT
$/* \ intra - consistency\ rules \ */$
...
$/* \ inter - consistency\ rules \ */$
$/* \ Model1 \ */$
$(Model1\_StageII\_Status = Finished \Rightarrow ...)$
$/* \ Model2 \ */$
$(Model2\_StageII\_Status = Finished \Rightarrow ...)$
...
OPERATIONS
Create_Model1_StageI = ...
Create_Model2_StageI = ...
Create_Model1_StageII = ...
Create_Model2_StageII = ...
...
END

Fig.2.7: General form of a B refinement for the subsequent Lyra stage

2.2.3 Model Based Testing for Lyra Models

The case study sets the requirements for the development of the model-based testing plug-in on the basis of experience gained at the Nokia research center. The requirements for this plug-in are described in D11 [2.9]. In this section we present preliminary methodology for model-based testing of Lyra B models.

Model Based Testing. Software models are usually built to reduce the complexity of the development process and to ensure software quality. A model is usually a specification of the system which is developed from the requirements early in the development cycle.

A software system built out of a formal model can be viewed as a refinement of the model. For correctness, any observed system behaviour must be a valid behaviour of the model. There are two ways of showing the correctness of implementation behaviour. One is to build a system through a succession of formal refinements and to demonstrate the consistency and refinement relationship at each step using the tech-

nique of *theorem proving*.

The other possibility is testing. Model based testing is different from code based testing: the former uses specification structure, while the latter uses the code structure [2.8]. A formal model can be subjected to symbolic execution to create a finite coverage graph; usually the coverage graph represents a state of the model and the edges denote operation invocations. One then can select a set of finite behaviours from this coverage graph based on a testing criterion. Each such behaviour can be seen as a test case; this approach is often termed as model based testing [2.3]. Testing is an incomplete activity; however, the test cases could be made effective in the sense that they capture the interesting aspects of the system and hence the success of their testing would give us confidence about the correctness of the system.

Model Based Testing from B models. B is a model-oriented specification language. By model-oriented we mean the system is modeled as an explicit state which can be modified by a set of operations. The behaviour of a B machine can be described in terms of a sequence of operations, and the first operation of the sequence originates from the initial state of the machine.

Model based test cases can be generated from a B machine in the following steps.

- *Step 1: Customization of the B machine:* The non-deterministic operations in a B machine will be customized by making them more observable; the System Under Testing (SUT) lets the test environment know of the choice it made in relation to a nondeterministic choice in the model. This helps in relating the implementation behaviour with the corresponding behaviour in the model.

- *Step 2: Creation of probe operations:* For each operation, we create a set of probe operations. These operations are either designed by the specifier or the tester. The results of these operations are used as oracle information. The use of the probe operations bridges the semantic gap between the abstract and the concrete name spaces.

- *Step 3: Creation of a signature file:* It is expected that the SUT has the same interface as that of the model. This means that all the B operations are present in the SUT and that each such operation has similar signature. The relationship between the operation parameters of the model with those in the implementation is defined using a mapping.

- *Step 4: Creation of operation instances:* The input space of a B operation will be partitioned into equivalence classes. Essentially, each partition corresponds to a distinct control path in the operation.

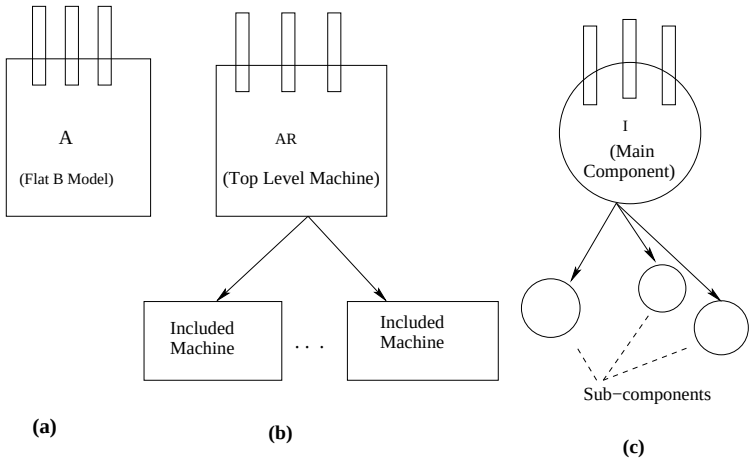


Fig. 2.8: **(a) Single B machine (B) Refinement of the same machine (c) Implementation**

- *Step 5: Creation of a Coverage Graph:* A coverage graph is created by performing symbolic execution over the B machine. A node in the coverage graph represents a state of the model and each edge is labelled with an operation instance invocation. A Testing criterion decides to what extent the coverage graph is created.

- *Step 6: Test case Generation:* The created Coverage graph will be traversed to generate sets of test cases. Each test case is a path in the graph starting from the initial state. The test cases are obtained in relation to the testing criterion.

Test case generation from Lyra-to-B models. During the first year of RODIN we have developed a set of formal specification and refinement patterns that can be used to systematically transform a Lyra model into the corresponding B model. In particular, we have introduced the notion of *Abstract Communicating Component* (ACC), which is a pattern for specifying a Lyra service component in B.

The instantiation of an ACC gives rise to a single B machine. In the subsequent phases of the Lyra in B method, this B machine is further refined to a hierarchy of B machines.

Since we have a strategy to generate test cases from a B machine, we can use the same method to obtain test cases for the B model obtained from the Lyra model. However, at this stage the test case generation strategy for B only handles flat B machines. Therefore, we will only generate test cases for the top level machine only. This could be seen from **Fig.2.8**. In the figure, (a) shows a flat machine called A; (b) is a refinement of the same machine in which the top module AR has the same interface as that of the machine A. In such a case, the test cases generated for machine A will also be test

cases of AR, the top module in the refinement. Let us assume the refinement in (b) is used to generate the implementation in (c) in which I is the top component which can call routines in the subcomponents. If the interface of I is the same as that of AR, then the test cases for AR could also be test cases for I. Thus the test cases of A are also test cases for I.

2.3 Demonstrators

The demonstrators for this case study will include:

1. the collection of formal B models (specifications) describing specification and development patterns for telecommunication systems,
2. a prototype of the tool supporting automatic translation of the Lyra UML2-based development process into specification and refinement process in the B Method,
3. a prototype of the model-based testing plug-in.

In this section we present formal B specifications describing specification and development patterns for Lyra models. The progress on 2) and 3) was presented in the sections 2.2.2 and 2.2.3 correspondingly.

2.3.1 Modelling a Service Component in B

We can define a Lyra service component as a coherent piece of functionality that provides its services to a service consumer via so called *Provided Service Access Points* (PSAPs). We use this term to refer to external service providers introduced at the Service Decomposition phase. However, the notion of a service component can be generalized to represent service providers at the different levels of abstraction.

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

While specifying a service component, we adopt a *systemic* approach, i.e., model the service component together with the relevant part of its environment, the service consumer. Namely, when modelling the communicational *(ACM)* part of *ACC*, we also specify how the service consumer places requests to execute a service in the operation *input* and reads the results of service execution in the operation *output*. The input parameters *param* and *time* of the operation *input* model the parameters of a request and the maximal time allowed for executing the service. The parameters of the request are stored in the internal data buffer in_data so they can be used by *ACAM* while performing the required computations.

In our initial specification we abstract away from the details of computations required to execute a service, i.e., *ACAM* is modelled as a statement non-determinis- tically generating results of service execution. These results are stored in the internal output buffer out_data. The service consumer obtains the results of service provision as the output parameter *res* of the operation *output*. Upon executing the operation *output*, the input and output buffers are emptied and the service component becomes ready to accept a new service request. Here we reserve the abstract constant *NIL* to model the absence of data.

MACHINE ACC

VARIABLES *in_data, out_data*

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

INITIALISATION
in_data, out_data := *NIL, NIL*

EVENTS

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

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

res ← *output* =
PRE ¬ (*out_data* = *NIL*)
THEN
res := *out_data* ||
in_data,*out_data* := *NIL*, *NIL*
END

END

In Lyra, a service component is usually represented as an active class with the PSAP(s) attached to it via the port(s). The state diagram depicts the signalling scenario on PSAP including the signals from and to the external class modelling the service consumer. Essentially these diagrams suffice to specify a service component according to the *ACC* pattern. Namely, the UML2 description of PSAP is translated into the communicational *(ACM)* part of the machine ACC. The functional *(ACAM)* part of ACC should be instantiated by the data types specific to the modelled service component. This translation formalizes the *Service Specification* phase of Lyra.

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

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

2.3.2 Formalizing Service Decomposition and Incorporating Fault Tolerance Mechanisms

In the first stage of our formalized development we used UML2 models produced at *Service Specification* phase to specify a service component according to the *ACC* pattern. The next step focuses on modelling service execution flow with incorporated fault tolerance mechanisms. Namely, we introduce a representation of *Service Director* into the abstract specification of a service component. This is done by refining the machine *ACC*. The result of this refinement – the machine ACC_DEC – is given below.

REFINEMENT ACC_DEC
REFINES *ACC*
SEES *Data*

CONSTANTS *Service, Eval, Next*

PROPERTIES ...

VARIABLES
in_data, out_data, time_left, old_time_left,
curr_task, resp, finished, results, curr_data

INVARIANT
...
(*finished* = **FALSE** $\Rightarrow$ *time_left*$>$0) $\wedge$
time_left $\leq$ *old_time_left* $\wedge$
(*finished* = **TRUE** $\Rightarrow$ (*resp=ABORT*) $\vee$ (*curr_task* = *size(Service)*+1))

VARIANT *time_left* + *old_time_left*

INITIALISATION
in_data, out_data := *NIL, NIL* ||
time_left, old_time_left := *max_time, max_time* ||
curr_task, resp := 1, OK ||
finished, results := **FALSE,** $\emptyset$ ||
curr_data := *NIL*

EVENTS

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

handle =
SELECT $\neg$ (*in_data=NIL*) $\wedge$ *finished* = **FALSE** $\wedge$ (*time_left* $<$ *old_time_left*)
THEN
old_time_left := *time_left*; *curr_data* $:\in$ *DATA*-$\{$*NIL*$\}$;
resp := *Eval*(*Service*(*curr_task*),*curr_data*);

CASE *resp* **OF**
EITHER *OK* **THEN**

results(*curr_task*) := *curr_data*;
curr_task := *Next*(*curr_task*);
IF *curr_task* = *max_sv*+1 **THEN** *finished* := **TRUE END**
OR *ROLLBACK* **THEN**
curr_task := *Next* $^{-1}$ (*curr_task*);
results := $\{curr\_task\} <<| results$
OR *REPEAT* **THEN skip**
OR *ABORT* **THEN** *finished* := **TRUE**
END
END
END;

timer =
SELECT $\neg$ (*in_data*=*NIL*) $\wedge$ *finished* = **FALSE** $\wedge$ (*time_left* = *old_time_left*)
THEN
CHOICE
time_left $:\in \{xx \mid xx \in$ **NAT1** $\wedge xx<time\_left\}$
OR
time_left, resp := 0, *ABORT ;*
finished := **TRUE**
END
END;

calculate =
SELECT $\neg$ (*in_data*=*NIL*) $\wedge$ *out_data* = *NIL* $\wedge$ *finished* = **TRUE**
THEN
IF *resp* = *ABORT* **THEN** *out_data* := *Abort_data*
ELSE
out_data := *results*(*Next* $^{-1}$ (*curr_task*))
END
END;

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

END

The machine ACC_DEC captures the design decisions made at *Service Decomposition* and *Service Distribution* phases. Namely, to derive the specification of *Service Director*, we use UML2 diagrams modelling both the functional and the platform-distributed architectures.

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

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

In the second case, the function *Next* describes the execution flow from the point of view of the main service director, i.e., it treats the groups of services managed by other service directors as atomic steps in the execution flow. For example, assume that the services S_1 and S_2 are managed by *Service Director1*, while S_3 and S_4 are managed by the $ServiceDirector2$. In this case the function *Next* treats the execution of S_3 and S_4 as one execution step whose performance is delegated to *Service Director2*. Hence, in this example *Next* will be instantiated as follows:

$$Next(S_i) = S_{i+1} \text{ for } i = 1, 2, \text{ and } Next(S_3) = S_5$$

The currently executed subservice is modelled by the variable $curr\_task$. The results of the current subservice are stored in the variable $curr\_data$. The results of all subservices already executed from the sequence 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 *Max_SRT*. The operation *timer* decreases the value of *time_left*, disables itself and enables the operation *handle*, which specifies the behaviour of *Service Director*. The variable *old_time_left* is used to force interleaving between progress of execution flow and the passage of time.

In the operation *handle*, we model not only requesting a certain subservice and obtaining its response, but also handling notifications about errors. We introduce the abstract function *Eval*, which evaluates the obtained response from a requested subservice. The result of evaluation is assigned to the variable *resp*.

If the subservice was successfully executed then the variable *resp* gets the value *OK*. In this case the next element from the sequence of subservices is chosen for execution

according to the function *Next*. If a benign failure has occurred and error recovery merely requires to retry the execution of the failed subservice then the variable *resp* is assigned the value *REPEAT*. 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. The inverse of the function *Next* defines which subservices should be re-executed, i.e., where in the sequence of subservices the execution flow should rollback to. 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*).

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

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

Let us observe that our approach to introducing fault tolerance can be seen as an abstract implementation of a rollback error recovery frequently used in distributed systems [2.4]. Indeed, the operation *handle* defines the rollback procedure by co-ordinating the error recovery according to the check-points defined by the function *Next*. The stable data storage is modelled by the variable *results*. The operation *handle* ensures consistency of the system state by the appropriate updates of *results*.

2.4 Future Development of the Case Study

Future work on the case study will proceed along the following direections:

- Modelling parallelism in the formalised Lyra development,
- Modelling dynamic reconfiguration in the Lyra service execution flow,
- Further development of the methodology for reasoning about fault tolerance in the presence of parallelism and reconfiguration,
- Automatic translation of Lyra UML2 models into B,
- Enhancing the model-based testing methodology for Lyra together with the work on the model-based testing plug-in.

We are planning to further develop the specification and refinement patterns to model parallel execution of communicating service components. The fault tolerance mechanisms already incorporated into the specification pattern for a service director will be enhanced accordingly. Also, formal modelling of dynamic reconfiguration in the Lyra service execution flow could become an especially challenging problem.

We are also going to continue our work on developing a formally verified profile for Lyra UML models. The developed profile will be used by a prototype tool supporting automatic translation of Lyra UML models into the corresponding B specifications.

The developed B models of communicating systems will be used to test and develop the methodology for model-based testing of Lyra B models. As a result, this methodology will be implemented into the model-based testing plug-in which takes automatically generated Lyra B models as sources for test generation.

References

2.1 R. Back and K. Sere. Superposition refinement of reactive systems. 1996.

2.2 Clearsy. AtelierB: User and Reference Manuals. Available at http://www.atelierb.societe.com/index_uk.html.

2.3 I. El-Far and J. Whittaker. Model Based Software Testing. 2001.

2.4 E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. ACM Computing Surveys, Vol.34, No.3, September 2002.

2.5 L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius, and Q. Malik. Formal Service-Oriented Development of Fault Tolerant Communicating Systems. TUCS Technical Report 764. Turku Centre for Computer Science, April 2006.

2.6 L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius, and Q. Malik. Formal Model-Driven Development of Communicating Systems. Proceedings of 7th International Conference on Formal Engineering Methods (ICFEM'05), LNCS 3785, Springer, November 2005.

2.7 S. Leppänen, D. Ilic, Q. Malik, T. Systä, and E. 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.

2.8 J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating Test Data from State Based Specifications. 2003.

2.9 Rigorous Open Development Environment for Complex Systems(RODIN), Deliverable D11, Definition of Plug-in Tools. online at http://rodin.cs.ncl.ac.uk/.

2.10 Rigorous Open Development Environment for Complex Systems(RODIN), Deliverable D8, Initial Report on Case Study Development. online at http://rodin.cs.ncl.ac.uk/.

2.11 Rigorous Open Development Environment for Complex Systems(RODIN), Description of Work. IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.

SECTION 3. CASE STUDY 2: ENGINE FAILURE MANAGEMENT SYSTEM

3.1. Introduction

This section of the D18 report summarises the developments in year 2 of AT Engine Controls (ATEC) case study "Engine Failure Management System" as part of the RODIN project.

The work on the case study supports the work of the following task from the Description of Work [3.10].

T1.2.4 Validate and verify formal specification to evaluate usefulness of formal methods

The work since the last interim report has been presented to the RODIN project in a series of internal workshops and presentations outlined below.

Presentation on work (Zurich Plenary,Sept 2005) and meeting
A summary of year 1 dissemination. Some collaborative work with Aabo. Identifying areas of work.

Presentation To EU (Brussels Oct 2005)
A brief outline of case.

Presentation on work (Aix workshop, April 2006)
A summary of the case study development was presented

The case study has provided contributions to the following Rodin deliverables in year two.

D18 (D1.4) Interim report (this report)
D22 (D7.3) Metrics report
D19 (D2.2) Methodology

3.1.1. Background

Rodin methods and technology such as UML-B have shown promise in tackling failure management domain concerns for ATEC such as closing the semantic gap (i.e., closer mapping of the problem domain to the design) and providing a reliable reusable process to meet the demands of a safety critical environment.
The first year provided the case material for interested Rodin partners to develop their technological approach and contribution to Rodin methods and tools. ATEC began learning of the methods and technology. The University of Southampton in cooperation with ATEC developed a generic model of the failure management system (FMS) based on a UML_B profile. A summary of Year 1 work including some early evaluation is described in the deliverables [3.13 and 3.14].

3.1.2. Overview of year 2 work

The second year work on the case material consists of contributions by ATEC, University of Southampton (Soton) and Aabo Akademi (Aabo). Their contributions are summarised as;

1. Pilot evaluation study (ATEC)
2. Generic feature-oriented specifications in FMS (Soton)
3. The requirements manager tool (Soton)
4. Classic refinement development of FMS (Aabo)

In year 2 ATEC intended to focus on the development of FMS behaviour in the generic model with University of Southampton but has now redirected its focus towards independent evaluation of technology using a pilot study. This has come about due to feedback from the EU reviewer's comments encouraging more industrial evaluation of Rodin technology and the need by ATEC to gain more experience in order to contribute effectively to model behaviour. The Pilot model is described briefly in section 3.3 and references to its contribution to methods and tools are made in section 3.2. The pilot study is a small subset of the original failure management case. ATEC was guided to the view that separate investigation of "UML" and "B" would be an effective way to evaluate, justify and contribute towards UML_B development and has explored the modelling process with these technologies using the RODIN methods and tools where available. It was envisaged that lessons learned from modelling would provide a contribution to the behaviour of the generic model later. The aim of the Pilot study was primarily to provide a more manageable vehicle to learn and evaluate the technology.

The University of Southampton has continued developing methodology supporting the generic model. Work at Southampton has proposed a prototype process for the V&V of a generic specification of this type, demonstrating stage (1): validate structural model using test data – and stage (2): verify system instance data against structural instance model. This was done using the existing UML-B tool and ProB model checker. The specification has been decomposed into *features* as the first step in an investigation of feature-based description, refinement and composition of generic specifications. This investigation will establish how to structure such feature-based transformations using the relevant mechanisms of the Event-B language: refinement, decomposition and generic instantiation.

A student project group at the University of Southampton has developed a plugin for UML-B, the *Requirements Manager*. This tool is a PostgreSQL-based repository of FMS instance data, with functions to input and verify instance data against the generic model, and to upload the data to UML-B for generation of a system instance UML-B specification. As a user-acceptance test of the tool, the V&V exercise of the previous paragraph has been performed with a full system instance dataset.

Aabo Akademi has been working on a classical refinement development of the FMS [3.15].The main result of developing the FMS by stepwise refinement in B is a set of formal templates for specifying and refining the FMS. The developed FMS is able to cope with transient faults occurring in a system of multiple homogeneous analogue sensors. The formal templates specify sensor recovery after the occurrence of transient faults and ensure the non-propagation of errors further into the system.

3.2. Major Directions on RODIN in Case Study Development

This section describes the contribution of the case study to methodology and platform and Plug in development in this second year. Contributions have been made by ATEC, University of Southampton, and Aabo Akademi. ATEC provides an additional evaluation in the D22 (D7.3) Assessment report deliverable.

3.2.1. Methodology perspective from Pilot Study (ATEC)

The methodological approach adopted by the case study is modelling in UML-B [3.11]. The exploration of the technologies of UML and B offered by the Pilot study allowed ATEC to evaluate each technology from their own perspective which is intended to contribute towards assessing the UML_B methodology and its tools. The two technologies are different and offer different benefits to UML_B. UML is essentially an object orientated modelling perspective and has evolved from modelling in various domains for different types of user whereas "B" modelling is not object orientated but is often used where the domain requires rigorous development. Rigorous development is often viewed as difficult to adopt by developers inexperienced in formal methods. Conversely development in UML is said to lack the formality required for precise specification. The study provides some evidence to test these assertions in addition to assessing technological contributions that might be useful towards developing UML_B methodology.

Evaluation of methodology by developing a "B" specification

The Pilot model is outlined in section 3.3. The approach taken to modelling in "B" was to consider the pilot subsystem from an event viewpoint. It was felt that this would provide an insight into using the emerging "Event B" methodology which is also part of the Rodin methodology [3.10]. The current tool set does not support Event refinement but evaluating the existing tools is still useful as the new versions will be based on them.
The Tools used in the study were ;

ProB
Which is a model checker [3.2] which has been used to provide verification and validation of models developed during the case study. It provides the facility to check execution of the state space of the model by checking for invariant violations and deadlocks and provides some refinement checking. It also provides an animation facility which allows dynamic execution of the model which is useful for validation purposes

B4free/Click 'N' Prove
This Prover checks the internal logic of the model for inconsistencies. It provides the facility to discharge automatic and interactive proofs.

In the "B" model the pilot subsystem was initially viewed as a black box whereby the output events and input events were identified. The first model illustrated the relationship between input and outputs events and the allowable sequencing of the output events. ATEC found that identifying the events and consequent output states was easy, given the small subset in the Pilot. All of the output state variables were set by one of a collection of alternative events. The different valid combinations of settings produced only a few events. However if a larger number of outputs were to be considered then this approach may be problematic, as identifying the events reflecting valid combination settings of all state variables would be extensive. Conversely allowing each output setting to have its own event could misrepresent the sequencing of output event combinations.

The events in the Pilot were illustrated dynamically by the execution of the model via the ProB animation facility. This facility was found to be particularly useful for validation of the model and was easy to use. Validation by this method gave assurance that the specification was what was intended. The inclusion of invariants in "B" modelling helped the verification tools to automatically check that the specification was consistent with their invariants however there is still a need to confirm that the invariant is correct particularly for a novice. Furthermore the creation of an invariant for all functional behaviour was not always achieved. It was felt that further development of the animator facility would be particularly useful to support validation of the formal methodology, e.g., allowing for automated execution scripts, highlighting states that have changed etc. One envisaged use of such a development would be to generate a validation suite of automated scripts which could be exercised and extended as the model was refined.

In order to avoid the difficulties of obtaining good abstract specifications where, in reality, requirements may be uncertain, a process of idealisation - de-idealisation was developed. The process is described in the next section and may contribute towards a potential methodology for development by novice users.

In general, refinement was introduced to each abstract model by considering what events the output events were dependant on. An event decomposition approach to refinement was undertaken, where events were decomposed into sub-events. However it was found that choosing what and how to decompose events is to some extent arbitrary and that a novice may benefit from some form of methodological process

guidance. The work being developed by Aabo on fault templates and by Soton on features approaches (see following sections) should contribute to this guidance.
For example in the first refinement, a 'validation' event was introduced which non deterministically changed the state of a test outcome, the output events selected were dependant on this outcome. The decomposition may alternatively have shown an output dependency on states from a range of validation events each with different levels of determinisms. This could have produced a more difficult architecture to refine. In general the view taken was to introduce determinism into the model gradually but this is a discipline.

The verification tools used in developing the model were found to be beneficial to the process and were used in different ways. The ProB tool was easy to use and useful to model check aswell as animate the model. The tool was also useful for identifying deadlocks in the model which is not possible using the Prover tool. Refinement checking using the Prover requires the modeller to provide a gluing invariant relating the abstract model and its refinement. ProB, on the other hand, does not require a gluing invariant as it effectively discovers the relationship between the abstract and concrete models through a mechanised search. When it works, this fully automatic refinement checking of ProB is very useful. However, methodologically it was found that having to construct a gluing invariant for the Prover provides the modeller with valuable insight into the design. The Prover tool was found to be less intuitive to use and in general proving was found difficult by the novice due mainly to inexperience with proving. Despite this, ATEC found the tool useful to indicate where a difficulty was in the specification and used this to change the specification (see 3.2.2 below)
and so achieve automatic proof obligations rather than interactive ones. The ease of proof was helped largely through the extensive use of superposition in the refinement chain.

In general, it was demonstrated that formal modelling and verification was possible by a novice with little formal training in using the current RODIN tools and methods. The implication being that the rigor of the UML_B technology which utilises "B" and uses the same verification tools will not necessarily be a barrier to a novice to formal methods. The model of the pilot was however simplistic and did not require many "B" constructs. For the wider case study the model would need to be developed to cater for a larger number of elements. This may involve changing the event operations to work on collections of inputs which requires more complexity and understanding of B. The generic model development will address larger number of elements.

3.2.2. Refinement processing - Idealisation – De-idealisation (ATEC)

The Pilot study developed a two stage approach to its "B" model development which may contribute towards a process methodology.

The idealisation stage involved introducing functionality to the model incrementally but without formal refinement. Each functional increment was added to the previous B machine to form a new machine. These abstract machines are regarded as idealisations as the increments were not proven refinements. The intention behind the process was to explore the basic functionality of the model fairly quickly without the overhead and inherent difficulty of establishing good abstract specifications and proof of refinements. The standpoint being to establish the intended basic behaviour of the

model before defining its operation more consistently and formally. This was felt a useful and natural approach where requirements may be uncertain or evolving and modelling requires some early assurance and flexibility. It is analogous to prototype development. Each machine did include some simple invariants and each was checked using the ProB model checker and animator.

The second stage was to revise the machines by translating them into formal refinements, what has been termed de-idealisation. In some cases not all the machines needed to be used in the formal refinement as some of the machine behaviour could be combined to simplify the refinement path and still capture the validated requirement. Furthermore the machines in the idealisation could now be reviewed in order to strengthen the model invariants, and architecture and so ensure consistency in the refinement chain. The ProB model checker and prover tool Click'N'Prove/B4 free tool were used to verify the model.

A special case of idealisation was also developed in order to assist in proving the refinement chain. Here sufficient abstract detail was added to the idealised specification to satisfy the proof obligations that were generated. In Event B when a new event added in a refinement this needs also to be recorded in its abstract as a *skip* operation. In this approach when new events were added, the abstract versions in previous levels were not *skip*, but a non-deterministic alteration to the variables at that level. The re-validation of the behaviour was only established for the current refinement level and not for the previous abstract levels.

With hindsight it became apparent that the previous level, now altered, may no longer describe the desired behaviour at that level and leads one to question the validity of this methodological approach. For example the "difftest" event, introduced in the abstract levels, could now unlatch some failures (which was not previously the case). This is because its action is a non-deterministic change to the result rather than simply a *skip* which would have no behavioural effect. In this case the abstract levels were still consistent with their invariants and model checked though would probably have failed a validation check on their animated behaviour. Since the proof of the refinement using this approach may simply be proving a refinement of invalid behaviour, this may be a pointless exercise, and could potentially introduce errors in further development of the abstract specification.

In order to address this issue, a slightly more constrained abstract version was considered by retaining any conjuncts from guards or conditions that were based on variables in the previous level. This restricts the effect of the event in a way that corresponds with the refinement. Hence it stands a better chance of being an acceptable specification for the refinement. Its consistency and validity can then be examined using the ProB model checker and animator. In our case, this method produced abstract specifications that were consistent with the existing invariants and valid but there is still uncertainty that that this will always be the case.

3.2.3. Feature Composition approach to requirements specification (Soton)

Having developed a generic entity-relationship model of the requirements [3.13] the next step was to specify the behaviour required by these entities. This was found to be difficult because there was too much detail to cope with in one go. The generic model was at a fairly low level of abstraction. This was manageable when considering only the structural relationships between functional entities. When adding behaviour it became more important to concentrate on one functional area at a time, i.e. to separate concerns as far as possible.

It was decided to consider the abstract behaviour of one feature at a time and then link the features to obtain complete model with behaviour. We define a *feature* as a part of the system requirement that can be described as a single *goal* where a *goal* is a succinct natural language abstract statement of requirement(s). A *feature* should be functionally coherent and loosely coupled with other features. For example, the abstract model presented in [3.1] describes the feature, *confirmation/recovery of failure*. Its goal is to endow the system with a resilience to isolated perturbations (*tolerate perturbations*).

3.2.3.1 Feature decomposition in Event-B

The primary decompositional refinement method we will use is that of Event-B [3.16]. Event-B simplifies the classical B language in two ways: (i) the unit of behaviour is the *guarded event*, where the guard is expressed as a predicate *G(e)* on the state *e*, and the event as a before-after predicate *R(e,e')*,and (ii) there is no syntax of modular structuring or inclusion.

In considering the decomposition of a an Event-B machine *M* into *N* and *P* say, we distinguish *external* from *internal* variables - from the viewpoints of *N* and *P* - where only the former may be changed by both component machines. An external event acts only on external variables. Further, the two external events in *N,P* acting on external variable *e* must have distinct names. In the refinements of *N* and *P*, external variables are restricted to a common *functional* refinement relation only, i.e. *h(f)=e* where *e* in *N,P* is refined to *f* in *NR,PR*. External event $E = G(e) \Rightarrow R(e,e')$ in *N* is thus refined automatically to $ER = G(h(f)) \Rightarrow R(h(f),h(f'))$ in *NR*.

In this scheme the decompositional refinement method is given by the commuting diagram **Figure. 3.1**. Start with machine *M*. Construct component machines *N,P* as abstractions over *M*. *N,P* are refined by *NR,PR* respectively, and each of the latter is refined by the final composition *MR*. *MR*, by construction, refines *M*.

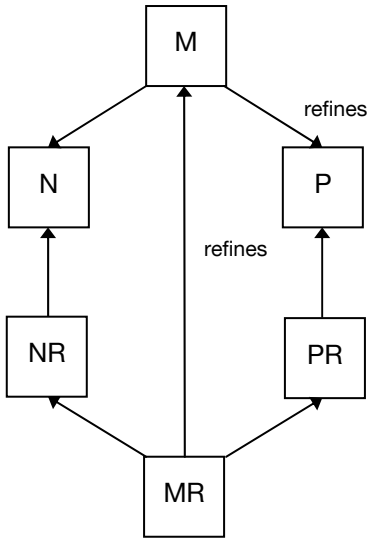


Figure 3.1 Decomposition of refinement in Event-B

In this scheme, a component (feature) machine N can only act on an external variable *v* in a prescribed manner: internal event *E* may act on *v* internally, and corresponding external event *Fe* models the effect of feature machine P's internal event *F* on *v*. The refinements of these events must be done subject to a common, functional refinement relation on *v* and its concrete counterpart *w*.

In order to embed a compositional feature-based approach in Event-B, we are investigating the extension of this form of decomposed refinement in at least the following ways:

- Two ``viewpoints" of some event *E*, i.e. two versions in two component machines, are combined in the recomposition of the refined components. A conjunctive combination, such as Butler's fusion operator [3.17], will be investigated.
- In recomposition, two component variables of distinct names and types, are identified through an equivalence relation. This would need to be expressed in a valid refinement.

Considering the strong constraint the refinement proof obligations represent, and the likelihood of inconsistencies arising in feature descriptions, or stakeholder views of these features, it is entirely possible that required refinement steps may not exist. The more liberal approach of retrenchment [3.18] has been proposed where refinement could not easily be applied in a feature interaction setting [3.19]; these ideas may be applicable.

3.2.3.2 Feature composition for the case study

The following functional areas (**Figure 3.2**.) were identified in the generic model.

1. *detection* of input failures
2. *confirmation*/recovery of failure
3. *action*s taken depending on status
4. interdependency *condition*s

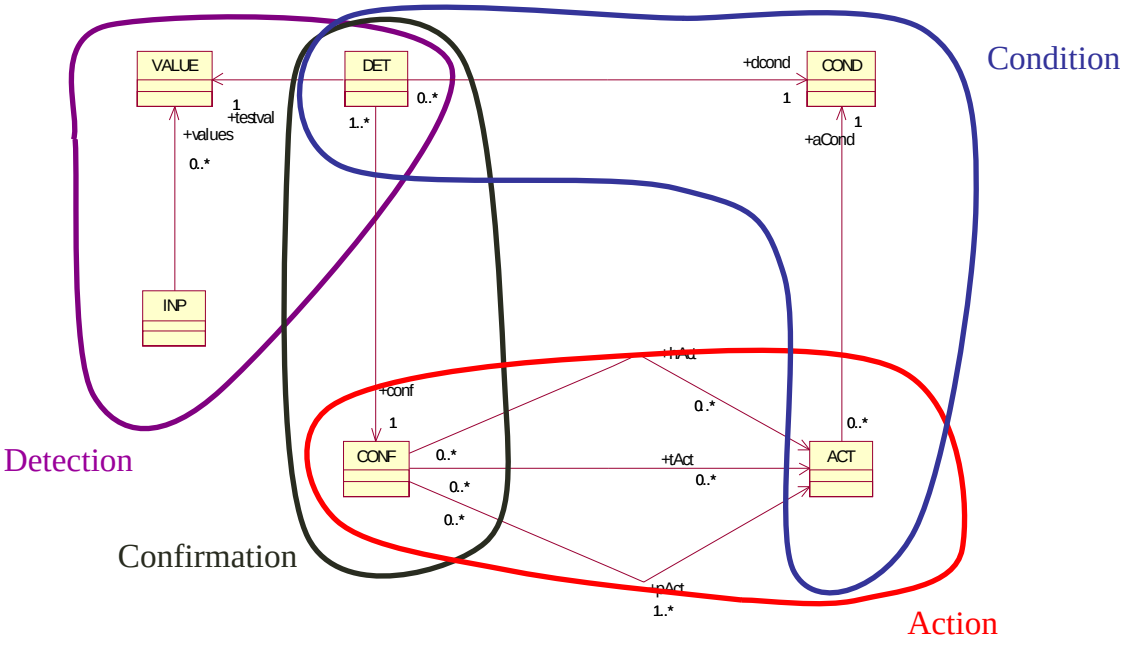


Figure 3.2. Functional areas identified in the generic requirements model

The feature corresponding to the functional area, confirmation, has already been identified in previous work. Features corresponding to the remaining functional areas are envisaged as illustrated in **Figure 3.3**.

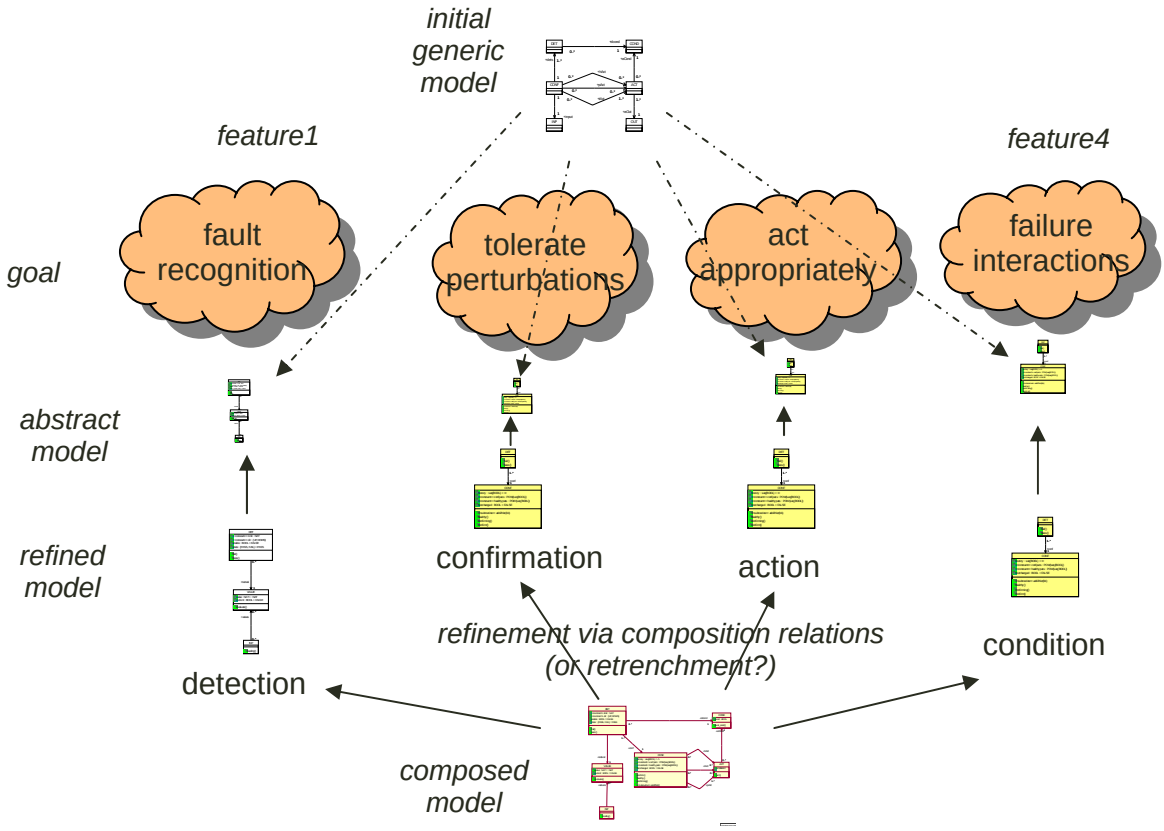


Figure 3.3. Refinement of abstract features to generate generic req'ments model

We envisage an approach consisting of the following steps.

1. Identify independent goals from the requirements specification.
2. Construct (and validate) via domain engineering [3.2] a structural class diagram of features and their relationships (generic model).
3. For each goal in the requirements spec:
    a. Construct a feature model of that goal by adding behaviour to the relevant part of the generic model
    b. Refine to add detail
4. Compose several features together to obtain original generic model with behaviour.
5. Show composition refines each feature model
6. Use ***composition relations*** (identified during feature modelling) to remove redundant abstract variables

Step 2 is necessary at an early stage so that independently developed features are based on the same underlying class structure. This assists when composing the features at step 4. Composition therefore consists of superimposing the properties of a class in one feature with those from the same class in another feature as shown in **Figure 3.4**. For composing the events of the class, it is first necessary to identify events that are essentially common, albeit differently named, to both versions. For these, events the guards and actions from both versions must be composed as shown in **Figure 3.4** to make a single event.

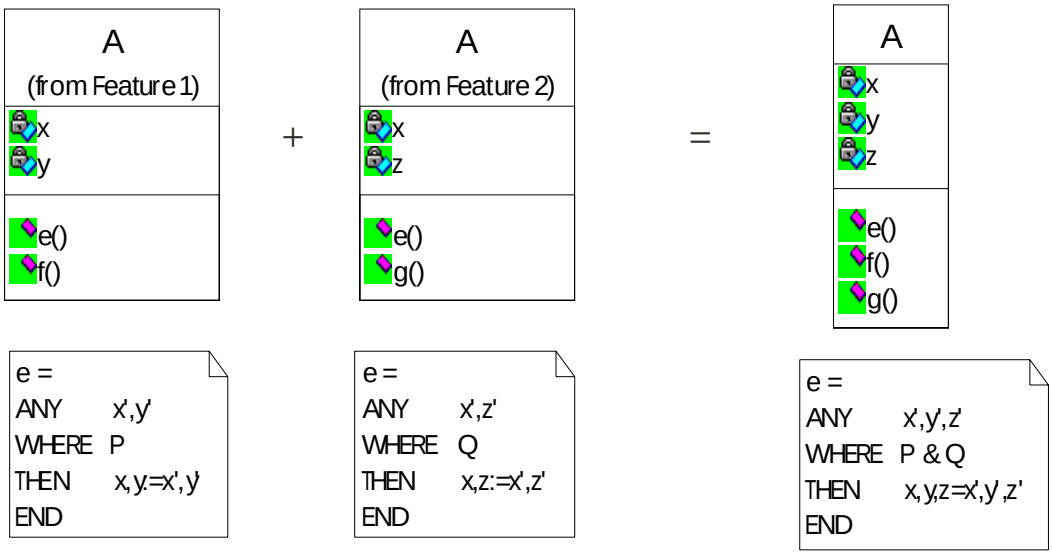


Figure 3.4. Composing the properties of the same class from two features

The superposition of properties from different features will lead to redundant data. In general, the redundant data will be an abstraction by one feature to avoid the detail contained in another. When the two features have been composed the correspondence of the two versions of the data can be established by adding and proving a composition relation as shown in **Figure 3.5**. The abstract version of the data can then be removed, and any references to it in guards or substitutions can be modified to use the more detailed representation as expressed in the composition relation.

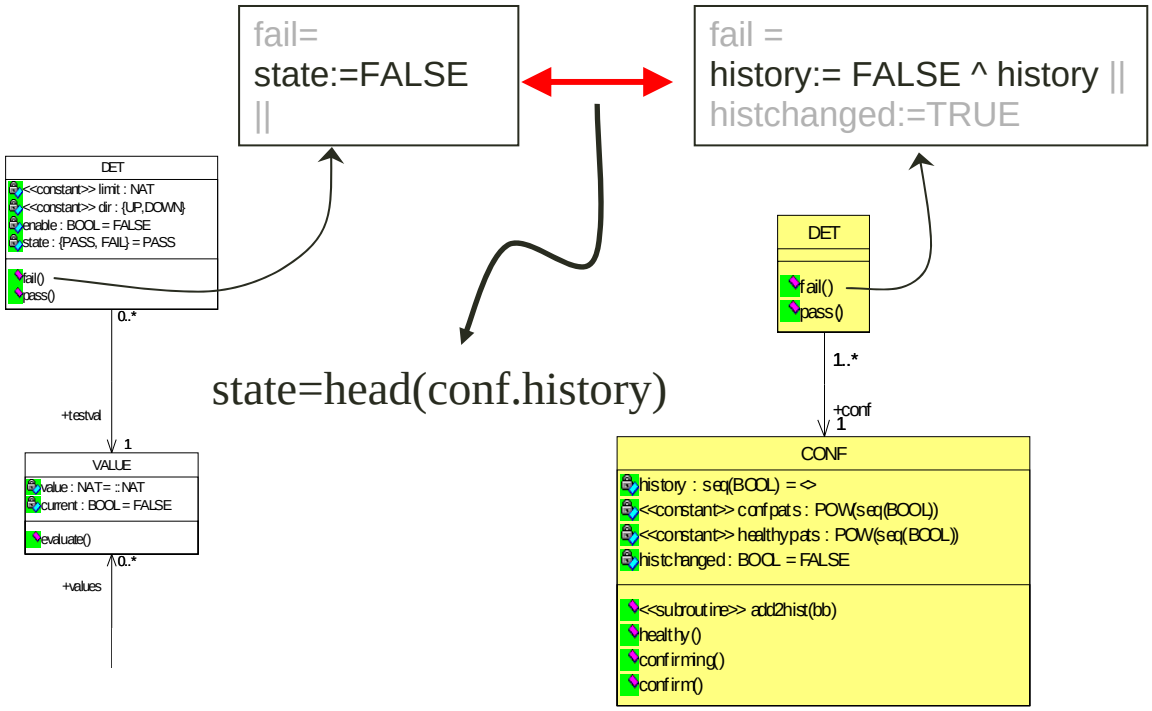


Figure 3.5. Identifying redundant data via a composition relation

The benefits of this feature composition approach to the construction of data intensive, configurable requirements models are,

1. Middle out (feature oriented) approach. Abstraction is difficult and, in practice, many people start in the middle before going upwards. Hence, this approach may provide an easier process to support refinement
2. Composition may be easier than decomposition. Our experience with industrial partners is that engineers automatically start composing systems from a set of components. This appears to be quite a natural approach to modelling for them.
3. Allows inconsistent requirements and makes compromise explicit. Refinement doesn't handle compromise very well and requires rework to ensure consistency. Although our approach is similarly based on refinement consistency, the isolation of functional areas into separate models may alleviate the problem by containing compromise rework within features.
4. Retains refinement route from requirements. The composed system is a valid refinement of each of several abstract feature refinements. Hence the full rigour of the refinement process is retained.

3.2.4. Product-line structuring and tooling (Soton)

Parnas's prescient early work [3.20] characterized three types of approach to the development of software product lines, or “program families” - (i) syntactic modification, (ii) modular specification, and (iii) refinement. At that time, type (i) involved the development of a complete program, followed by the production of variants by modifying the original program. Since then this type of approach has been elaborated through process phases e.g. requirements, architecture, and through the structuring of artefacts of those phases. All types of approach involve a domain engineering activity that captures the requirements that all family instances must share – the *commonalities* - and the requirements that vary between instances – the *variabilities* [3.22] - into a generic, reusable software resource. This is followed by an application engineering activity that uses this resource to generate the specific instance systems as necessary. Most product line work assumes an early domain analysis activity, e.g. [3.21], for gathering and structuring all relevant information from the application domain to support the development of such a generic, reusable software resource.

It is noteworthy that the Parnas's approach type (i) remains dominant today. In this type of approach, application engineering deploys an instance derivation process against generic models/architectures and specific components/interfaces and variation points, to generate an instance system: an elaborate process of syntactic modification. Some logic-based and formal methods techniques have also been proposed for software product lines. In particular, formal refinement-based approaches (Parnas's third type of approach above) largely remain to be applied to software product lines.

In this case study to date we have illustrated a product-line approach to the rigorous engineering, validation and verification of structural generic requirements for critical systems such as failure management and detection for engine control. The approach should be applicable to any system composed of multiple instances of similar units - specified using parameterised reuse - where the extent of variability and dependency between units makes such reuse non-trivial to achieve. The major part of the work remains – to extend the product line model to incorporate variant sets of behaviours.

Our approach in this work is of Parnas's type (i). Initially, a class diagram of the generic product line model is created. On input of data for a system instance, the RM database checks that the data satisfies the dependencies of the class diagram, and facilitates user “debugging” of erroneous data. The system instance is specified by “populating” the UML-B stereotype with this verified instance data, and U2B then combines the generic and instance information into an instance B specification, for further formal verification with ProB and theorem Provers. What distinguishes our methodological and tooling work for product lines is its integration with a leading language and method for formal specification, refinement and verification, Event-B.

3.2.5. Methodology of developing the FMS using formal specification templates (Aabo)

An acute issue in developing the FMS is design of the mechanism for tolerating and recovering from transient faults of the system components. In [3.15] we presented an approach to developing the FMS with the mechanism for tolerating transient faults by classical refinement in the B Method.

The formal development of the FMS starts with an abstract specification defining the behaviour of the FMS during one FMS cycle. The stages of such a cycle are:

- obtaining inputs from the environment,
- performing tests on inputs and detecting erroneous inputs,
- deciding upon the input status,
- setting the appropriate remedial actions,
- sending output to the controller either by simple forwarding the obtained input or by calculating the output based on the last good values of inputs,
- freezing the system.

At the end of the operating cycle the system finally reaches either the terminating (freezing) state or produces a fault-free output. In the latter case, the operating cycle starts again.

In our abstract specification we model the readings of N multiple homogeneous analogue sensors as inputs to the FMS. After obtaining the sensor readings, the FMS starts the error detection. At this level of abstraction we model only the result of error detection, which can be either TRUE if an error is detected on the sensor reading on a particular input, or FALSE otherwise.

Based on the results obtained at the detection stage, the FMS non-deterministically decides upon the status of an input (i.e., a particular sensor), which may be classified as *fault-free*, *suspected* or *confirmed as failed*. *Suspected* inputs are those faulty inputs which still may recover. The remaining faulty inputs are designated as *confirmed as failed*. Upon completing analysis, the FMS applies one of the corresponding remedial actions: it either forwards a fault-free input to the system controller, calculates the output based on the information about the last good input value, or enters the freezing state.

The refinement process of the FMS starts by elaborating on the input analysis procedure. As a result, the input analysis is performed gradually by considering inputs one by one until all the inputs are analyzed. It is based on the results of error detection and the values of the input status obtained at the previous cycle of the FMS. Namely, if an analysed input was previously *fault-free*, it becomes *suspected* after an error is detected. If the input was already *suspected* and an error is detected again, it can either stay *suspected* or become *confirmed as failed*.

The procedure for determining the input status can be further refined by introducing a customisable counting mechanism which re-evaluates the status of a particular input at each cycle and also allows the system to distinguish between recoverable and unrecoverable transient faults.

Further development of the FMS continues by refining the error detection procedure. The mechanism of error detection relies on a specific architecture of error detection actions called *evaluating tests*, which may vary depending on the application domain. The basic category of the evaluating tests is called *simple* tests. A simple test is executed based solely on the input reading from a sensor. After all simple tests associated with a certain input are executed, so called *complex* tests can be performed. The results of complex tests depend on the results of the associated simple tests.

Since the FMS works with homogeneous multiple sensors, for each of N sensor readings the same series of tests is applied. The tests are executed considering one input at a time, until all the inputs are tested. For each input, we select the tests to be executed according to certain requirements. Namely, each test can be executed at most once on a certain input. If the test is complex, then all the associated simple tests have to be executed before it. If an error on an input is detected then no more tests on that input should be performed. The result of the execution of each enabled test is modelled non-deterministically. If the result shows that the test on the input failed, the input is found in error.

Every test is executed with a certain frequency. The test frequency together with the additional condition on the internal system state determines the enableness of a test for execution. To model the execution of tests according to the given frequencies, we introduce *time scheduling*. The real time is modelled by introducing the event which gradually increments the current time value. The progress of time is only allowed when one FMS operation cycle finishes and before the next one starts, or when there are no tests enabled for execution under given conditions. In the latter case, we allow time to progress and possibly the internal system to be changed state until some tests become enabled. After executing all required tests on a particular input, the FMS classifies the input as found in error or error-free.

The detailed description of the FMS behaviour while performing error detection and input analysis is given in the form of the B specification templates (see the section on demonstrators). These templates can be instantiated to develop a domain-specific FMS.

3.3. Progress on the Demonstrators

The initial intention was to develop a verified and validated visual model of the failure management system for demonstration and then develop and demonstrate a generic system from which variants can be derived. These activities are being developed in parallel.

Development of non generic model behaviour specific to failure management is being addressed by Aabo with their work on fault templates. Soton have been addressing generic systems issues with their work on features, product line and requirement tool development.

Although ATEC's work in the Pilot study has largely been concerned with learning and evaluation of technology, the pilot model is outlined here and demonstrates a

simple example of FMD behaviour and how its modelling has been addressed by a novice user.

3.3.1. Overview of Pilot model (ATEC)

The definition of the engine failure management system as a subsystem has been described in the deliverable of D2 [3.6] and in the initial presentation of the project. A dual sensor input (Esa and Esb) for engine speed (Es) was chosen from case study 2 requirements document to model as part of the Pilot study. It was chosen as it included common behaviour representative of other control inputs and also included some interaction between its dual inputs.

The Es output is normally derived from the Esa input but if this is not healthy (ie. the failure management system detects a failure) then a healthy Esb input will be used if available. If both inputs are not healthy the Es signal is not updated and a freeze flag set. A context diagram for the Pilot is shown in **Figure 3.6** below.

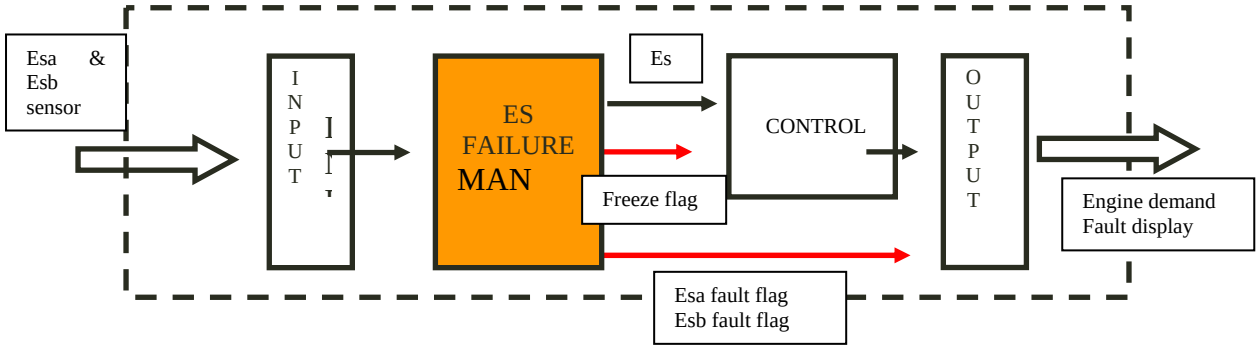


Figure 3.6 Es Context

The main functionality to be modelled included

1. an magnitude in range test
2. a difference test between two inputs
3. a confirmation mechanism

3.3.2. B Model development Engineering

The approach taken to modelling the pilot has been overviewed in sections 3.2.1 and 3.2.2. Further description of the model development is provided in the REFT book [3.3].

Five refinements were undertaken and are informally described below;

Table 3.1 Informal Refinement description

Abstraction	Description
R0	The pilot subsystem was viewed as a black box whereby the output states would be latched when failed. The combinations of fault flag and control value ES were output from the box in response to a change in Esa or Esb inputs.
R1	This refinement introduced a dependency of the output on the results of a validation event i.e. if the output state is not latched its outcome will depend upon the state of the validation. The validation is still non deterministic.
R2	This refinement introduced sequencing to the events by strengthening the event guards to further constrain and refine the order that events may occur.
R3	This refinement introduced an event comparing the difference between the dual input values
R4	This refinement introduced determinism into the events i.e. the magnitude inrange testing and confirmation mechanism. This also resulted in an increase in output events

In the intial abstraction (R0) the output of the system was viewed as only having 4 states where the combination of esalatch and esblatch variables can be set to either "FAILED" or "HEALTHY" represented in the abstraction as event operations hh,hf,fh,ff. A sanitized excerpt of the abstract specification is shown in **Figure 3.7** below.

```
MACHINE Engine speed 0

... INVARIANT ... & (newVal = TRUE or
                     (esalatch = UNSET & output = esavalue) or
                     (esblatch = UNSET & output = esbvalue) or
                      output = previous)

OPERATIONS /*EVENTS*/
esaChange=
BEGIN
     esavalue :: NATURAL || newVal := TRUE
END;

hh =
SELECT
       esalatch = UNSET & esblatch = UNSET & newval = TRUE
THEN
       output , previous := esavalue, esavalue ||
       newVal := FALSE
END;
 ... ff =
BEGIN
```

```
        output := previous ||
        esalatch, esblatch ,freeze:= SET, SET,SET ||
        newVal := FALSE
END
```

Figure 3.7 - R0 Base Machine

The abstract specification shows how the invariant maintains the output states. Changes to the input are illustrated by an event sequencing control flag "newval" indicating that an input has been read.

The specification under specifies several requirements e.g. input and output range and granularity, timing relationship of changes of input and output events, it also relies on certain assumptions such as the input being able to be processed quick enough in order to determine output states.

The next refinement R1 looked in more detail to the events that contributed towards the output states in keeping with the Rodin event refinement approach. This refinement introduced a dependency of the output on the results of a validation event i.e. if the output state is not latched its outcome will depend upon the state of the validation. The validation is still non deterministic. A sanitized excerpt is illustrated in **Figure 3.8** below.

```
REFINEMENT Engine speed 1
REFINES Engine speed 0 ... ;

esavalidate =
BEGIN
esaresult :: PASS FAIL
 END;

hh =
SELECT
        esaresult = PASS & esbresult = PASS &
        esalatch = UNSET & esblatch = UNSET
THEN
        output, previous := esavalue, esavalue ||
        newVal := FALSE
END;
```

Figure 3.8 – R1 refinement

The final refinement R4 greatly reduces non determinism. A sanitized excerpt is illustrated in **Figure 3.9** below.

```
Gluing invariant
invariant=
```

```
….
((esaresult_R4 = PASSED)=>(esaresult = PASS))
& ((esaresult_R4 = FAILED)=>(esaresult = FAIL))

/*single input evaluation*/
esavalidate=
SELECT esavalidated=FALSE THEN
   IF esalatch=PASS THEN
         IF ((esavalue<esallmt)or (esavalue>esamlmt)) THEN
            IF esarange_ct>actmlmt  THEN esaresult_R4:=FAILED
            ELSE esarange_ct:=esarange_ct+actinc
                  ||esaresult_R4:=CONFIRMING
             END
         ELSE IF esarange_ct>0 THEN
                  esarange_ct:=esarange_ct-actdec
                  || esaresult_R4:=PASSED
               ELSE  esaresult_R4:=PASSED
        END
   ELSE esaresult_R4:=FAILED
   END
  || esavalidated=TRUE
  END
/* dual input evaluation*/
esdiff=
SELECT esdiffvalidated=FALSE THEN
   IF esaresult_R4 =PASSED and esaresult_R4=PASSED THEN
         IF (esavalue>esbvalue+diflmt) THEN
            IF esdiff_ct>3  THEN esbresult_R4:=FAILED
            ELSE esdiff_ct:=esadiff_ct+difinc
                  ||esbresult_R4:=CONFIRMING
             END
         ELSE IF esdiff_ct>0 THEN
                  esdiff_ct:=esdiff_ct-difdec
                  || esbresult_R4:=PASSED
               ELSE  esbresult_R4:=PASSED
        END
   END
  || esdiffvalidated=TRUE
  END


 Example of New event
cF =
 SELECT
    esalatch=UNSET & (esblatch=UNSET  or esbresult_R=FAILED)
    & newval=TRUE
    & esavalidated=TRUE & esbvalidated=TRUE   & esdiffvalidated=TRUE
    & esaresult_R4=CONFIRMING
THEN
     o_ES:=pre_o_ES || esblatch=SET
```

```
    Newval:=FALSE||esavalidated=FALSE||esbvalidated=FALSE
    ||esdiffvalidated=TRUE
END
```

Figure 3.9 – R4 refinement

The refinement has introduced determinism to the validation event. The adopted confirmation mechanism introduced a new validation state "CONFIRMING" which resulted in the creation of several new output events. Of course the new events resulting from this new state could have been considered in the abstract machine R0 at the beginning but it was felt that introducing the change as a later refinement would show how the model and process can handle change and be useful for evaluation purposes. It was felt more realistic, as requirements may change as the model develops.

The model raises several issues for consideration in the demonstrator model.

1. The model has been decomposed in a particular way. It has relied on certain environmental assumptions and under specifies some behaviour.
2. Confirmation and detection behaviour has been combined, this may be more maintainable in a generic model if decoupled
3. There is a high degree of flag setting to control sequencing of events
4. There is some redundancy in the model as the testing of latch setting is already performed in the validation events and does not need to be addressed in the outputs
5. There is some dependency on events eg difference testing dependant on validation of single inputs. Aabo's work on test dependency addresses this issue.
6. How easily can the model be scaled up for to include other control inputs? Eg. would simply introducing events associated with each given output accurately depict the total subsystem behaviour? Would it lead to inefficient architecture?

3.3.3. Configuring the generic model for the Failure Management Requirements using a tool for instance data management (Soton)

To address the problems found with using ProB to verify instantiation data (see Initial report on case study [3.13]), we developed a tool that interfaces with the UML drawing tool to automate management and verification of instance configuration data. The tool was developed as an IBM Eclipse plug-in by a student group. The tool provides an extension to the Rational Software Architect UML modelling tool (also based on Eclipse). Menu extensions are provided to operate the tool from the class diagram so that a database repository can be generated based on the classes and their associations. Class instance and association link data can then be 'bulk uploaded' directly from the Excel configuration files containing the specific requirements data. This avoids the tedious and error prone process of manually populating the class diagram with this information. A small sample of the requirements data is shown in

Figure 3.10.

TABLE:INP				TABLE:CONF				TABLE:OUT			
	ref				*ref*				*ref*		
	ESa				conf_ESa				freeze		
	ESb				conf_ESb				cES		
					conf_ESdiff				fESa		
TABLE:DET									fESb		
	ref	*'dcond*	*'conf*	TABLE:COND					fESd		
	ESa_upper	always	conf_ESa		*ref*						
	ESa_lo_starting	starting	conf_ESa		always			TABLE:HACT			
	ESa_lo_runnning	running	conf_ESa		starting				*ref*	*'conf_h*	*'hact*
	ESb_upper	always	conf_ESb		running				HACT_R1	conf_ESa	use_main
	ESb_lo_starting	starting	conf_ESb		idling				HACT_R3	conf_ESb	use_backup
	ESb_lo_runnning	running	conf_ESb		notHardFaulted						
	ESa_rate	always	conf_ESa		ESafaulted			TABLE:PACT			
	ES_diff	idling	conf_ESdiff		notESafaulted				*ref*	*'conf_p*	*'pact*
					notESbfaulted				PACT_R1	conf_ESa	use_lgv
TABLE:INP_R					ESbfaulted				PACT_R3	conf_ESb	use_lgv
	ref	*'det*	*'inp*								
	INP_R1	ESa_upper	ESa	TABLE:ACT				TABLE:TACT			
	INP_R2	ESa_lo_starting	ESa		*ref*	*'acond*	*'out*		*ref*	*'conf_t*	*'tact*
	INP_R3	ESa_lo_runnning	ESa		use_main	always	cES		TACT_R1	conf_ESa	set_fESa
	INP_R4	ESb_upper	ESb		use_backup	notESbfaulted	cES		TACT_R2	conf_ESb	set_fESb
	INP_R5	ESb_lo_starting	ESb		use_higher	always	cES		TACT_R3	conf_ESb	freeze
	INP_R6	ESb_lo_runnning	ESb		use_lgv	always	cES		TACT_R4	conf_ESdiff	use_higher
	INP_R7	ESa_rate	ESa		freeze	ESafaulted	freeze		TACT_R5	conf_ESdiff	set_fESd
	INP_R8	ES_diff	ESa		set_fESa	always	fESa				
	INP_R9	ES_diff	ESb		set_fESb	always	fESb				
					set_fESd	always	fESd				

Figure 3.10. Requirements data

Some types of verification errors (such as mismatches between the class diagram and tables and referential integrity errors) may prevent the data from being uploaded. **Figure 3.11** shows the error view provided by the tool. Several such failures are shown in the lower half of the error view.

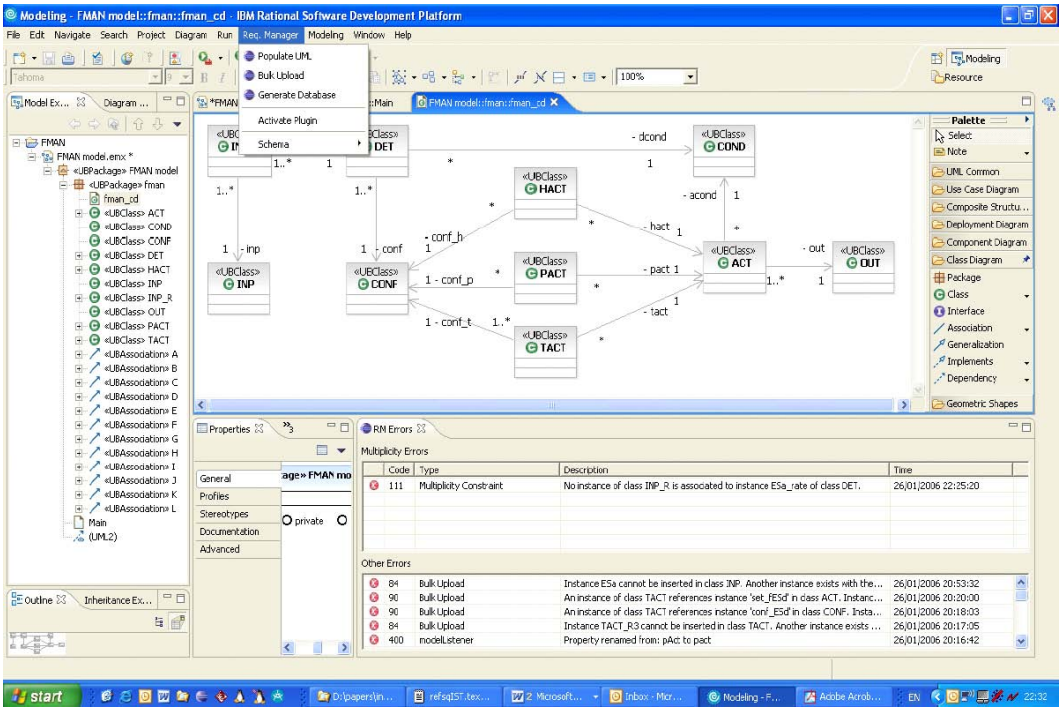


Figure 3.11. Tool screenshot with error view

If the configuration data has none of these errors it is loaded into the database schema and further verification is performed by checking class diagram constraints, such as multiplicity constraints on associations. These multiplicity errors are shown in the

upper half of the error view in **Figure 3.11**. The error messages identify the particular data instances that violate the multiplicity constraint giving sufficient information to pinpoint the problem. The error can then be corrected either by editing and re-loading the configuration data, or by editing the database from the class diagram. For the latter, an extension to the class pop-up menu is provided, giving direct access to the relevant database table as shown in **Figure 3.12** where a multiplicity error is being corrected. Although the tool identifies the nature of the error more precisely than ProB (by giving all counter examples whereas ProB only identified which constraint was violated), it may still be difficult to find the correct solution. In the example in **Figure 3.11**, it is clear that ESa rate detection is missing an associated input but it may not be obvious which link in the table needs replacing. Knowledge of association links throughout the class diagram are needed to find a correction. In future work we intend to provide tools to visualise the transitive association links for a given set of class instances by automatically generating object diagrams from the database.

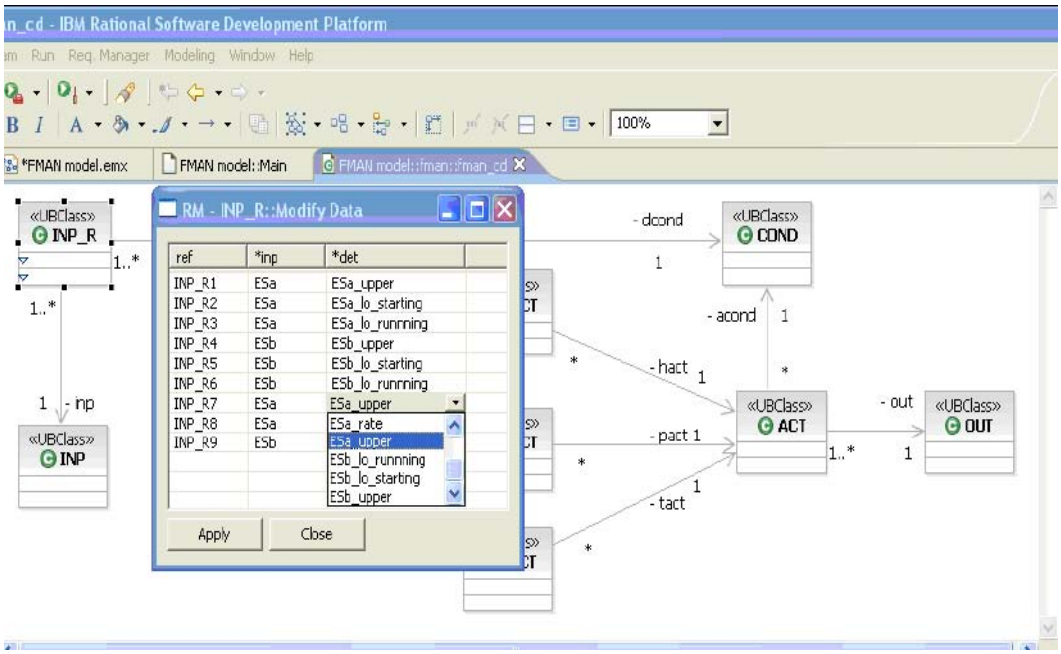


Figure 3.12. Tool screenshot with data editing window

The ability to modify data via the class diagram enables individual class instances and association links to be added to an existing (or developing) configuration. The requirements engineer can invoke the Requirements Manager(RM) tool at chosen stages (when the data is expected to be in a consistent state) to check the configuration satisfies the generic constraints of such systems.

A limitation of the database approach to managing configuration data is that many to many association relations can not be represented in database schema. In order to represent many to many associations intermediate linking classes are added to the class diagram (see INP R inserted between INP and DET and HACT, PACT and TACT between CONF and ACT). In future versions of the tool we intend to hide this representation mismatch from the user.

The RM tool has been developed as an Eclipse plug-in to integrate with the RODIN project toolset including the UML-B drawing tool, U2B translator, ProB, B Prover and B database. In parallel with the development of RM, the U2B tool has been re-developed in Eclipse to accept input models based on the UML2 metamodel (upon which RSA models are based). A UML2 profile has been developed to extend the UML notation and provide relevant property fields to accept information such as the configuration data. Once the configuration data has been successfully verified RM can be used to populate the UML-B stereotype properties. This utilises the *instances* property attached to classes and the *value* property attached to associations. These values are utilised by U2B when it produces a B version of the model. **Figure 3.13** shows the stereotype property *value* for an association after population by RM.

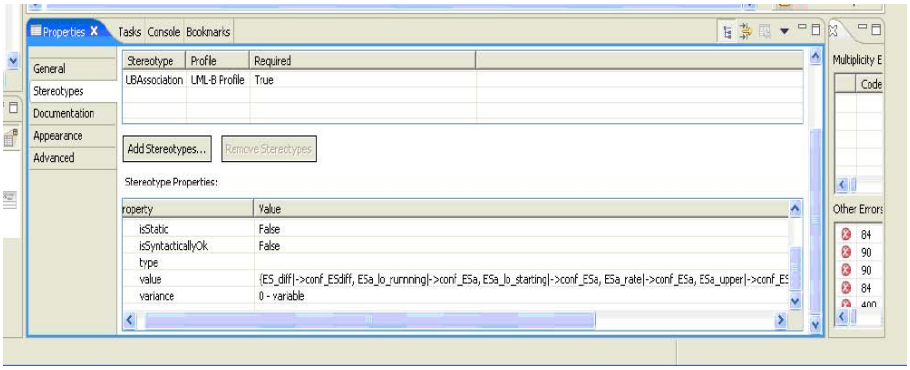


Figure 3.13 Tool screenshot showing *value* property after populating with RM tool

3.3.4. Development of the FMS with specification and refinement templates in UML (Aabo)

The formal development of the FMS can be enhanced by the use of the widespread graphical modelling language – UML. The approach taken is motivated not only by wide acceptance of UML in the industry but also by the existence of a tool (U2B [3.5] which translate informal UML models to their formal B counterparts.

The development of the FMS is performed in phases. Each development phase is characterized by the set of UML models (class and statechart diagrams) depicting the main structural and behavioural aspects of the FMS at a certain level of abstraction.

The **1st development phase** models a very abstract FMS cycle:

1. the FMS reads input values from the sensors, then
2. it performs some abstract (i.e., unknown at this phase) action, and then
3. it either calculates the output or fails.

If the output is successfully calculated, the FMS cycle starts again.

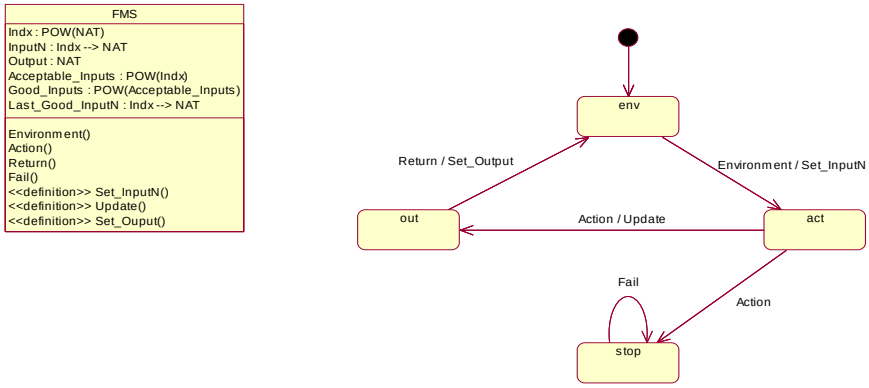


Figure 3.14. FMS Class and fms_state Statechart diagram for the 1st FMS development phase

The UML template for this first development phase is given in **Figure 3.14**. The class 'FMS' outlines the structure of the system, where the class attribute 'Indx' is a set of sensor indexes, InputN is an array of input values, and Output is a system output. In addition, Acceptable_Inputs is a subset of sensor indexes for inputs which have not yet failed (i.e., are either error-free or suspected), Good_Inputs is a subset of sensor indexes for error-free inputs, and Last_Good_InputN is an array of the last good input values. The statechart diagram fms_state attached to the class FMS describes the FMS behaviour via a set of the FMS states and transitions between them.

In B, the class FMS corresponds to a B machine, the variables of which are obtained from the attributes of the class FMS. The B operations model the class methods corresponding to the transitions of the statechart diagram.

Using the U2B tool, the B specification can be automatically generated from the above UML diagrams. For instance, the FMS cycle starts with executing the Environment() method. The action triggered (Set_InputN) models reading of the input values. As a result, it arbitrarily sets the values of the attribute InputN. The B operation corresponding to the method Environment() is as follows:

Environment = SELECT fms_state=env THEN fms_state:=act || Set_InputN END;	where Set_InputN is defined as: *Set_InputN* == InputN :: Indx → NAT

More generally, each method of the class FMS has the following form:

Method = SELECT fms_state=outcoming_state THEN fms_state:= incoming_state || triggered_Action END;	where triggered_Action is a method of the FMS class with the stereotype <<definition>>.

Each subsequent FMS development phase refines the structure and the behaviour of the original system. For instance, in the **2nd FMS development phase** we introduce the input analysis performed after obtaining the sensor readings.

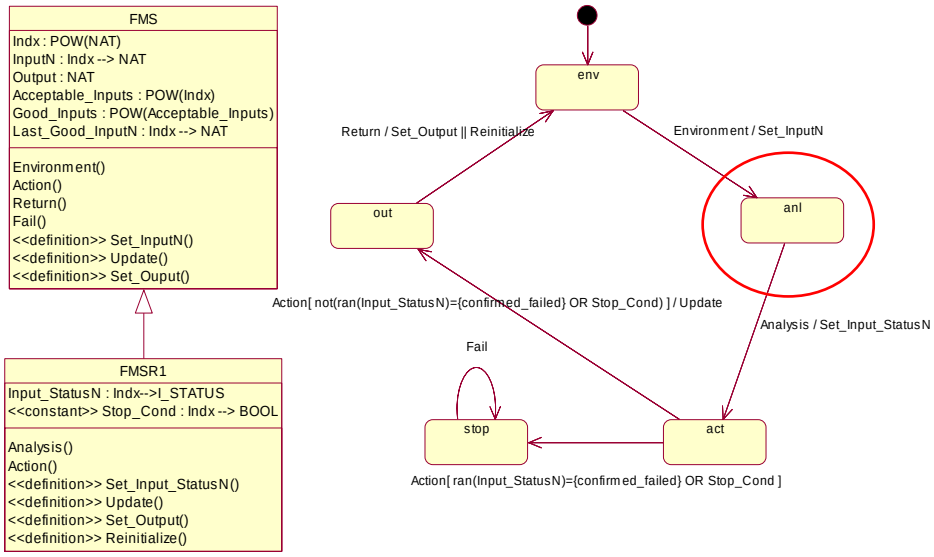


Figure 3.15. FMSR1 Class and fms_state Statechart diagram for the 2nd FMS development phase

The initial UML template is refined as shown in **Figure 3.15**. Namely, the FMS is now represented as the class FMSR1 that refines the class FMS from the 1st development phase. In our semantics it means that FMSR1 realizes FMS. As a convention, in the realization class, only new attributes and methods are shown, as well as those methods which are changed. Refining the FMS behaviour requires introducing a new state in the fms_state statechart - anl. In B, this corresponds to data refinement of the variable fms_state by extending its underlying type with the new element anl. The new transition from this state defines the method for the refined FMS functionality – Analysis(), which classifies the inputs obtained from the sensors into three categories: *ok*, *suspected* or *confirmed_failed*. Correspondingly, the result of the analysis is modelled as the additional attribute Input_StatusN.

In the **3rd FMS development phase** we introduce an abstract representation of error detection that the FMS performs after obtaining the sensor readings. The FMS is now represented as the class FMSR2 (given in **Figure 3.16**) which refines the class FMSR1 from the 2nd development phase. The newly introduced attribute Input_In_ErrorN models the results of error detection. The behaviour of the FMS is refined by introducing the new FMS state det and the method Detection(), which performs error detection on inputs and, as a result, classifies them either as erroneous or error-free.

The **4th FMS development phase** specifies in more detail the input analysis in the FMS. At this phase, we focus on analyzing inputs one by one based on the results of error detection. The UML template for this development phase is shown in **Figure 3.16**. The FMS is represented as the class FMSR3 which refines the class FMSR2 from the 3rd development phase. To model gradual input analysis, we introduce the looping state anllopp into the existing statechart. While in this state, the FMS analyzes the inputs one by one and saves the intermediate analysis results. To ensure that the analysis loop terminates, we introduce the additional attribute Processed that keeps the information about the already processed inputs. Once all the inputs have been analyzed (processed), the FMS cycle can proceed and the appropriate actions can be applied.

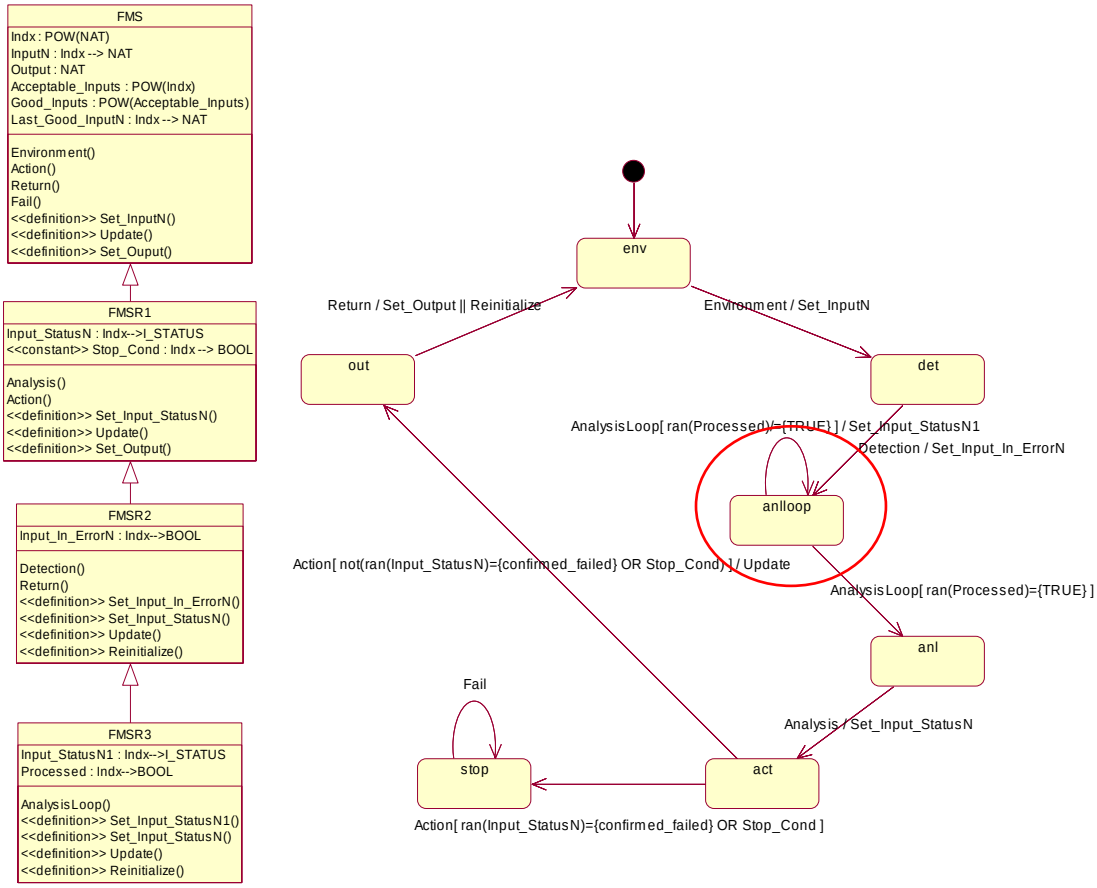


Figure 3.16. FMSR3 Class and fms_state Statechart diagram for the 4th FMS development phase

The **5th FMS development phase** specifies in more detail the FMS error detection by introducing input tests. At this phase, we focus on detecting errors (applying tests) on inputs. Again, the inputs are processed one by one and the intermediate results of error detection are saved. After performing error detection, the FMS cycle can proceed by starting input analysis. The template for adding a looping state for gradual error detection is similar to the one presented in the previous phase.

3.4. Future work

In year 2, work has continued into the Failure Management Domain by building on first year work and developing further domain engineering methods. Some evaluation of the emerging technology has been undertaken which has given some credence to the usability of the methodology by an industrialist who was a novice to B and to formal methods. The study identified the need for more guidance and flexibility in the model development processes together with greater emphasis on methods to help with validation of requirements in developing models. The work by the academic partners is contributing to this.

The University of Southampton has successfully developed the requirement manager tool in response to Year 1 which identified the need to provide assistance with configuring the generic model. Further work on feature development and Product line

engineering is providing a process framework for developing generic models in the future. The emphasis here, on composition rather than decomposition, may contribute towards future guidance for reusability. Future work using UML-B inheritance and refinement to elaborate system behaviour will relate the approach to Parnas' type (iii) - refinement. For example, the abstract model now specifies generic detection behaviour in terms of checking an input-derived value against a limit. A refinement specializes this behaviour for magnitude, rate, multiple etc. detection types. Thus we anticipate that the instance data population stage will become more elaborate, populating a graph of refinements, rather than a single model as at present. The Event-B method under development by the RODIN project will provide mechanisms for composition and decomposition with refinement, to support scalable development. The generic instantiation mechanism of Event-B, whereby a model can be made generic with respect to one or more configurable contexts, will afford a component-like form of reuse.

To progress this agenda, current Soton project work is examining:

- the extensibility of the RM tool to more expressive data constraint specification (i.e. invariants) over instance data. Currently RM models type-correctness, and association constraints via referential integrity. This scheme should be extensible in the manner of tools such as DECIMAL [3.23].
- the possibility of providing data visualisation/ exploration facilities for RM instance data for data debugging purposes.
- cost/benefit analysis of moving RM to the Eclipse EMF/GMF metamodelling/ graphic tool framework, which at this early stage appears to offer capabilities of fast model construction and expressive constraint specification.
- potential for exploitation of existing feature modelling plugin technology for Eclipse.
- possible extensions to UML-B both from UML notations not yet incorporated, and other Requirements Engineering frameworks such as van Lamsweerde's KAOS [3.24].
- possible extensions to UML-B to enhance the specification of complex event sequencing in required behaviours. Existing work integrating B and UML statecharts with CSP will be considered here.

Aabo have been addressing modelling behaviour in the FMS domain by developing model templates. Some already provide solutions to observations raised from the pilot model ie patterns to handle test dependency and scheduling. Further development will contribute towards demonstrators of modelling in the domain and provide templates that could guide development.

The next FMS development phases for Aabo are:

- introducing special counters to ensure termination of error recovery,
- introducing time scheduling into error detection,
- modelling different types of tests executed in the error detection procedure.

The overall methodology results in both UML templates and automatically generated B machines, correctness of which can be verified using the B Prover.

ATEC work in the final year will be divided between evaluation of the technology which will now include newly Rodin developed platform and tools eg Event b, and

UML_b and collaboration with the academic partners. The evaluation work will continue along the lines of the pilot study by evaluating UML_B using the new Rodin toolset. Collaboration work will consist of some independent working with both partners. This will include development of guidelines to model development which is also expected to be influenced by ATEC evaluation work on RTCA certification guidelines for aviation software. ATEC hope to start adding behaviour to the generic model by using the experience of the pilot study.

A Summary of work being considered for the final year includes:

1. To contribute to Demonstrators
   - UML_B development with generic pattern (Soton,ATEC).
   - UML _B development templates for FMS (Aabo,Soton).
2. To contribute to methodology
   - Elaboration of generic modelling & product line methods (Soton).
   - Guidelines for developing UML_B models (Aabo,Soton,ATEC).
3. To contribute to evaluation
   - Considerations of compatability with DO-178/ED-12 standard for Airbourne software (ATEC). This work will involve comparing formal development in UML_B against meeting the requirements of certification standards for airbourne software.
   - Evaluation of new Rodin methodology and tools (ATEC).
4. For further consideration
   - Prototype Implementation of FMS (ATEC).
   - Traceability of requirements in modelling and refinement (Aabo).

3.5. References

[3.1] C. Snook, M. Butler, A. Edmunds, and I. Johnson. Rigorous development of reusable, domain-specific components, for complex applications. In J. Jurgens and R. France, editors, *Proc. 3rd Intl. Workshop on Critical Systems Development with UML*, pages 115–129, Lisbon, 2004.

[3.2] C. Snook, M. Poppleton, and I. Johnson. The engineering of generic requirements for failure management
Accepted for Eleventh International Workshop on Requirements Engineering: Foundation for Software Quality, REFSQ'05, Oporto, 2005.

[3.3] C. Snook, M. Poppleton, and I. Johnson. Towards a methodology for rigorous development of generic requirements
To appear in Proceedings of Workshop on Rigorous Engineering of Fault Tolerant Systems, REFT, Newcastle, 2005.

[3.4] C. Snook, I. Oliver, and M. Butler.The UML-B profile for formal systems modelling in UML. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems*, chapter 5. Springer, 2004.

[3.5] C. Snook and M. Butler. U2B - A tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[3.6] RODIN deliverable D2 : Definitions of Case Studies and Evaluation Criteria Project IST-5111599, November 2004.

[3.7] RODIN deliverable D4 : Traceable Requirements Document for Case Studies Project IST-5111599, February 2005.

[3.8] RODIN deliverable D9 : Preliminary Report on Methodology IST-5111599, September 2005.

[3.9] RODIN deliverable D14 : Assessment report 1 IST-5111599, September 2005.

[3.10] Rigourous Open Development Environment for Complex Systems -RODIN :Description of Work IST-5111599, April 2004.

[3.11] C. Snook and M. Butler, *"UML-B: Formal modelling and design aided by UML", To appear in ACM Transactions on Software Engineering and Methodology*, 2006.

[3.12] J.-R. Abrial. *Mechanical Press*: Requirement Document, November 2004.

[3.13] RODIN deliverable D8 : . Initial report on case study developments IST- 5111599, September 2005.

[3.14] RODIN deliverable D14 : D7.2 Assesment report IST-5111599, September 2005.

[3.15] Dubravka Ilic, Elena Troubitsyna, Linas Laibinis and Colin Snook. *Formal Development of Mechanisms for Tolerating Transient Faults*. TUCS Technical Report, No.763, April 2006.

[3.16] Metayer, C. and Abrial, J.-R. and Voisin, L. RODIN deliverable D7 : D3.2 Event-B Language, IST-5111599, May 2006.

[3.17] Back, R.J.R. and Butler, M. Fusion and simultaneous execution in the refinement calculus, *Acta Informatica, 35:11*, pp921-949. 1998.

[3.18] Banach, R. and Poppleton, M. Retrenchment: An Engineering Variation on Refinement, In Proceedings of Bert, D (ed.) *2nd International B Conference: Recent Advances in the Development and Use of the B-Method, LNCS Vol.1393*, pp.129-147, April 1998.

[3.19] Banach, R. and Poppleton, M. Retrenching Partial Requirements into System Definitions: A Simple Feature Interaction Case Study, *Requirements Engineering Journal, 8:4*, pp266-288, 2003.

[3.20] Parnas, D. L., On the Design and Development of Program Families, *IEEE Transactions on Software Engineering, SE-2, March 1976*.

[3.21] Prieto-Diaz, R., Domain Analysis: An Introduction, *ACM SIGSOFT Software Engineering Notes 15:2* pp.47-54 1990.

[3.22] Coplien, J. and Hoffman, D. and Weiss, D., Commonality and Variability in Software Engineering *IEEE Software, November/December 1998*, pp.37-45.

[3.23] Padmanabhan, P. and Lutz, R. Tool-supported verification of product line requirements, *Automated Software Engineering 12:4* pp. 447-465, Kluwer Academic, October 2005.

[3.24]A.vanLamsweerde
Goal-Oriented Requirements Engineering: A Guided Tour
Invited Paper for RE'01 - 5th IEEE International Symposium on Requirements Engineering, Toronto, August, 2001, pp. 249-263.

SECTION 4. FORMAL TECHNIQUES IN MODEL DRIVEN ENGINEERING CONTEXT

4.1 Introduction

We summaries here the developments of Case Study 3 - "Formal Techniques in MDA Context" during the second year of the Rodin project. The primary goal of this is to investigate how the techniques and tools developed within Rodin can be applied in a heterogeneous, model based environment and work flow.

To evaluate this we are applying the Rodin tools and techniques to the development of a (primarily) hardware based mobile phone platform known as NoTA which provides a Corba/ Web Service environment to "services".

This work has been presented internally within Rodin at the Zurich and Aix-en-Provence workshops in September 2005 and April 2006 respectively.

Results of this work published and/or presented externally include:

- Abo Akademi Technical Report 759 on the use of fault tolerance patterns for state machines [4.2]
- Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis, May 2006 [4.8]
- A paper on the semantics of UML's notion of composite structure at Systems Analysis and Methods conference, May 2006 [4.7]
- Synthesising Hardware using B and Bluespec, to be published at Forum on Design Languages'06 in September [4.5]
- Keynote speech on the "Use of Formal Methods with UML in Industry" at Nordic Workshop on UML'06, Grimstad, Norway.

In addition, work from Rodin has also appeared in the following papers

- A Method for Mobile Terminal Platform Architecture Development, to be published at Forum on Design Languages'06 in September [4.6]
- A UML Profile for Asynchronous Hardware Design at the SAMOS IV workshop and also accompanying LNCS volume. [4.1]

Furthermore, use of Rodin technology has appeared in 3 patent applications during the past six months. Pending submission also for the B conference in 2007 in both the academic and industrial tracks.

Section 4.2.3 is written by Vesa Luukkala and parts of section 4.2.2 by Pontus Bostrom, with contributions from Mads Neovits, Marina Walden and Ian Oliver.

4.1.1 Tasks

In the Rodin Description of Work Document a number of tasks are described for this case study. We summarise these here and describe briefly the current status.

T1.3.1 Define Case Study

As will nearly all industrial projects, the scope of the project changes over time as the project matures. The Mobile Internet Technical Architecture is a proposed framework for mobile device and ecosystem structure. Within this context the NoTA project is one of the first major ideas and implementations to arise from and to be based upon this work. In this respect we can consider NoTA to be fulfilling all the requirements of the MITA plan.

The major evaluation criteria are based primarily around the suitability of the Rodin tools and technologies to be used in our current and future development processes. At this stage we are confident that this is the case. The tools within Rodin at this time are less mature than some of the technologies and this will be pursued to a greater degree in the final year of Rodin.

T1.3.2

As described later in this document the use of the Rodin technologies was directly employed in parts of the NoTA requirements definition process with the result of the NoTA High Interconnect Specification. Work is currently underway in the implementation of this specification in both software and hardware forms. This work should be completed by the end of 2006 and a detailed comparison of the defects in the formally developed versions versus the earlier NoTA prototype versions be made.

T1.3.3

Much of this work was made in earlier projects with the U2B tool in particular. We have extended some of this to other parts of the UML such as composite structure.

T1.3.4

NoTA has been a continually changing project and the specification has reflected this over time. The version described in this document is effectively the third major incarnation of the description of the NoTA system. If there is a major result to be announced now (we are actually waiting until after the HIN release, T.1.3.2) then it is that the formal specification has reduced the amount of requirements change over the course of the project.

T1.3.5

This has not been approached at this time but it is hoped that collaboration can be made with at least Ambient Campus

T1.3.6

See T1.3.2 and T1.3.4

T1.3.7

This is scheduled for the final year of the project.

4.2 Major Directions in Case Study Development

This case study is directly related with Nokia's Network on Terminal Architecture work. This project is an ongoing piece of work to develop the "next generation" of mobile terminal platforms based upon the ideas of service orientation

In figure 1 we show the interplay between the case study and its current foci. In particular we have been concentrating more on the development of parts of the NoTA system and from this we have then had a structure on which other aspects of the case study can be investigated.

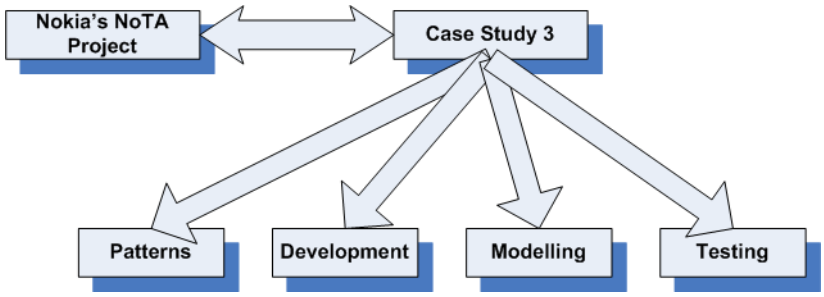


FIGURE 1. Overview of Casestudy Foci

In the following sections we describe the work that has been made in each of these areas and finally how this relates back to the real project at Nokia. These can be classified as in figure 1 as follows:

- Semantics of UML Structures (Modelling)
- Fault Tolerance Patterns for State Machines (Patterns)
- Model Checking and Animation (Development)
- Model Based Testing in NoTA (Testing)
- Hardware Description Language Generation (Patterns, Development)
- NoTA High Interconnect v2 Specification (Development)
- Modelling Framework (Modelling)

Regarding the task of petri-net based modelling, this has not be covered in any detail during the past year and it is likely that this area has no real application to the work here due to changes in our focus over the period of the project. Some related work in this area however has been made but outside the strict scope of Rodin [4.1].

In the following section we describe the above as follows: Firstly we present a discussion of the composite structure notion in UML and the results found analysing its usage there with a short discussion on how this is to be incorporated in the existing UML to B mappings. An overview of the fault tolerance pattern with state machines is then made. This is followed by a section describing the animation, CSP testing and its relationship to the theorem proving of the NoTA High Interconnect layer (H_IN). A section on model based testing and how that area is being constructed is then made followed by a section on how B and a hardware description language are being integrated. Finally the H_IN v2 specification section describes the design of the H_IN layer from the domain modelling, through architecting and then the choice of the B language to continue the specification of this later.

At this point in time we do not wish to present the modelling framework as we do not feel that it is of a mature enough state to discus in a formal manner.

4.2.1 Semantics of UML Structures

This work has arisen from the need within this case study to expand the usage of UML from the traditional class diagram structures to the more esoteric and less used diagrams such as the composite structure diagram. There appear to be a number of ways of working with the composite structure and also a lack of concrete semantics either regarding the meaning of the diagram (or artifact) itself and also its relationship with other UML elements. We undertook an analysis of this and described the use of this diagram or artifact type in terms of the UML class diagram - which can obviously be processed by the U2B tool from Southampton; this has the advantage that to use the composite structure diagram we do not have to update the mechanics of the U2B tool but can rely on the existing translations.

There have been two distinct areas of work here, one related with the UML semantics itself and one on transforming one model into another through the application of patterns. In this section we concentrate on the semantics of UML and how this might be utilised by the U2B tool.

The composite structure diagram's and related structures' uses and semantics are well described in while the notions of composition are adequately described in. Its function is to extend the modelling capabilities of the UML beyond that of classes and their relationships and is primarily aimed to assist the modelling of the internal structures of classes with a more well de ned notion of decomposition. Similar notions exist in methods such as ROOM (capsules) and languages such as SDL and SysML for example. An example of the composite structure diagram is given in figure 2.

It is important to recognise and understand how constructs such as the composite structure are being used in reality and how close (or far) these usages are from or to the inventors' aims. If these facts are understood then the development of languages such as the UML becomes more relevant to the current practise. We have collected this body of results over a number of years of working with internal consulting projects utilising the UML and its various profiles within Nokia. The range of projects has been from enterprise database systems to embedded, real-time components.

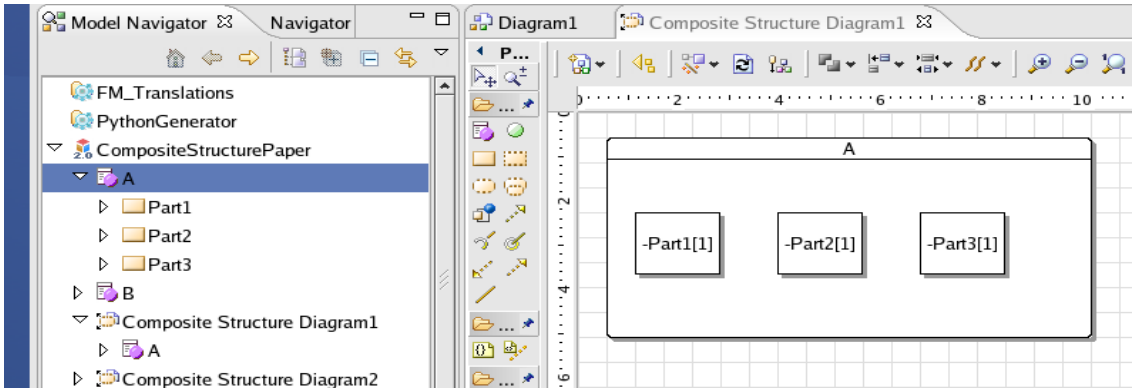


FIGURE 2. Composite Structure Diagram in Borland's Eclipse Based Modelling Tool

In our work we have identified a number of common ways of utilising the composite structure diagrams in conjunction with the class diagram for describing various systems. While the UML2.0 Superstructure document describes the syntax and provides some informal semantics we have found it necessary to either enhance these descriptions and in some cases extend the meaning in order so that this particular UML artifact is used consistently and effectively. In this section we describe the uses of composite structure diagrams and their interpretations:

- Architecture Specification
- Scenario Descriptions
- Modes
- Operation Effects

Of these we only consider Architecture Specification and Scenario Descriptions here.

By far the most common usage of the composite structure is for architecture specification. In a number of tools, for example, Borland's Together Architect, the composite structure is used to describe the configuration of a class and associated parts. For example, figure 2 shows a composite structure diagram viewing a model. This model contains two classes A and B, where class A has three parts (named Part1, Part2 and Part3). The composite structure diagram shows class A's parts in completeness.

This is per the UML2.0 semantics and each composite structure diagram acts as a view [20] onto the internal structure - the parts - of a given class; this view may omit entities that are inside the given class. The issue here is that the composite structure diagrams do not stand alone as being artifacts of the model and that the diagram does not necessarily show a complete view of the internal structure - this is something that is often misunderstood and not just with these particular diagrams but also with the class diagram.

For the practitioner, a common interpretation of a composite structure diagram, as currently implemented in the UML2.0, is that it describes the system as a whole or the 'architecture of the system.' A note must be made about the definition of 'architecture': while definitions do exist, it is often unclear to the practitioner (and sometimes even to the persons developing the methods) what architecture really is. Suffice, we often find that the term is misapplied to mean a top-down, functional decomposition of objects which omits the architectural definition or development step that requires human imagination and inventiveness to adhere to design requirements in favour of a mechanical process of simple decompositions.

This usage is often in conjunction with the concept of a 'system' class much in the same was as a top-capsule is used in ROOM or as the block interaction diagram is used in SDL. Almost invariably in these cases the class diagram (if ever expressed) appears as a top-down decomposition structure with an entry or start point in that system class.

A further usage of the composite structure is where the diagram or diagrams are used to describe the various configurations of the system. In a process which then is similar to describing scenarios with object diagrams as a way of requirements elicitations and then reconstructing the class diagram. Various composite structures are combined and generalised to produce the class diagram which admits all the given composite structures as shown in figure 3.

The interaction between class and composites structure has a subtle effect for the modeller: it is recommended that multiplicity constraints, the type (aggregation, relation, composition) and directionality are always denoted on associations between classes. However this sometimes is difficult to envisage when working from a class perspective and it requires scenarios to be modelled using object diagrams and such multiplicity constraints and other properties to be inferred from these scenarios.

This particular way of working can be difficult if not foreign, especially as the drawing object diagrams are not supported by many tools (we have used USE for this) and that most OO practitioners are not trained to utilise object diagrams. This style of using composite structures is not covered by the UML2.0 but can be 'admitted' by some tools1.

But the collaboration, at least to the modeller, does not appear as a diagram in its own right but as an artifact that requires a diagram - this might explain why modellers are reluctant to use it.

The collaboration artifact or element is not read in the same way as a composite structure diagram, the usage is more akin to that described in. By 'read' here we mean in the semiotic sense such that persons used to engineering the structure of systems interprets and writes the descriptions of those systems with the tools (in our case the composite structure diagram) that they are used to thinking with. Anecdotal evidence suggests that engineers think of modes as the system comprising of objects in a certain structure and not as a collaboration of objects, where as the collaboration is used to describe particular scenarios and not modes of operation. One can also argue that an additional or yet another UML construct to learn is also complicating matters.

Now this all relates to the U2B tool and mapping work in that we can show a mapping between the composite structure and the class diagram in a formal manner that is not present in the UML syntax and semantics. Figure 3 below shows a partial relationship between the two diagrams.

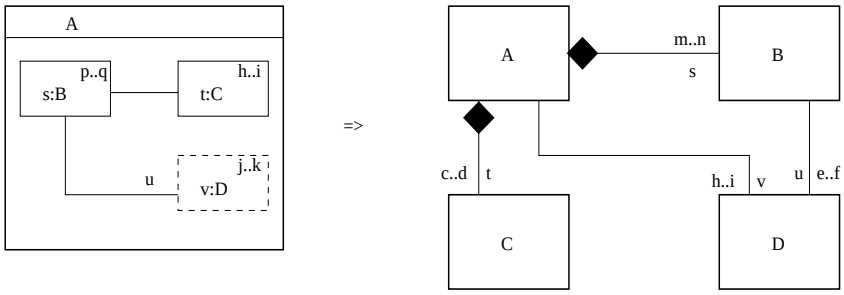


FIGURE 3. Relating Composite Structure and Class Diagrams

As U2B primarily works on mapping the class structure to B, a demonstration of the relationship of the various other diagram types (in particular the composite structure) to the class diagram makes these admissible for translation using the current schemes. This obviously saves on the mapping work required by the developers of U2B and secondly provides a much stronger internal semantics for the UML.

There does remain an issue about the precise semantics of the composition relationship (that is the black diamond symbol above) but current experiments show a further simple mapping which will be described in a later report.

4.2.2 Fault Tolerance Patterns for State Machines

A platform independent model (PIM) in a Model Driven Architecture (MDA) context considers only features in the problem domain. In order to implement the platform independent model, the model is transformed into a platform specific model (PSM) that takes into account implementation issues for the platform where the system will run. The PSM is not necessarily a refinement of the PIM, since it can introduce features that are not considered there at all. For example, fault tolerance and other platform specific features should not be included in the PIM, since every possible platform where the system could run would have to be taken into account. All potential platforms might not even be known at the time the PIM is created.

In order to anticipate all the different restrictions that will be encountered on a specific platform, the fault handling mechanisms and other platform specific features in the PIM would have to be very general. Hence, they would not provide any useful information and could restrict future transformations to other platforms. We introduce an automatic transformation of the PIM to allow a very abstract definition of fault tolerance and other platform specific features. These platform specific features can then be refined to concrete features in the platform specific model.

We use UML to describe the platform independent and platform specific models. Here we concentrate on state-machines; we do not consider the object oriented features of UML. To have a formal semantics and good tool support for analysis, the state-machines are translated to Event B. Event B is a formalism based on Action Systems and the B Method for reasoning about distributed and reactive systems. It supports stepwise refinement of specifications and it is also compatible with UML state-machines.

Even if the PSM is not necessarily a refinement of the PIM, we would still like to preserve as many refinement properties as possible in the transformation from PIM to PSM. The following properties of the PIM are required to be preserved in the PSM:

1. The sequence of valid calls to public operations is maintained in the PSM or the state-machine has reached state exit.
2. New public operations (UML events) are not introduced. Hence, an object does not require new interactions from its environment.
3. New behaviour violating the refinement relation between the PIM and the PSM cannot take control forever.
4. There should be a trace in the PIM that is also a possible trace in the PSM. Hence, it should be possible to execute the PSM using only the transitions in the PIM.

The rules above can also be expressed as restrictions on the state-machine in the PSM. In the view of the environment, a UML state-machine accepts a language over an alphabet consisting of the events. Assume that state-machine in the platform independent model accepts the language L. The alphabet of the language is the public operations of the object. Consider two strings L1 and L2 in L such that L=L1L2. In order to introduce fault tolerance, we can add new error handling mechanisms. Assume the error handling mechanism is represented by the string of events f and the error can occur between L1 and L2. The state-machine of the platform specific model can then accept the following language L1L2+L1fL2+L1f. This means that the state-machine operates either normally, recovers and continues with its normal operation or it terminates.

To enable transformation of the PIM to a PSM we use anticipating events in Event B. We transform the PIM to a model having all the possible anticipating transitions modelling very abstract fault tolerance features.

In figure 4 we illustrate how a platform independent model M is transformed into a platform specific model M". First M is automatically translated to a model T(M) including all the possible anticipating events. The model T(M) is hidden from the developer of the PIM. He/She will only have to consider the models M, M' and M". To obtain a model M' with platform specific features, a pattern p1 is applied to the PIM M by the developer. This procedure can be repeated until all platform specific features have been introduced. The result obtained is a model that has similar functionality as M, but can have several platform specific features, e.g. fault tolerance. The obtained model M' is a refinement of T(M), but not necessarily of the platform independent model M.

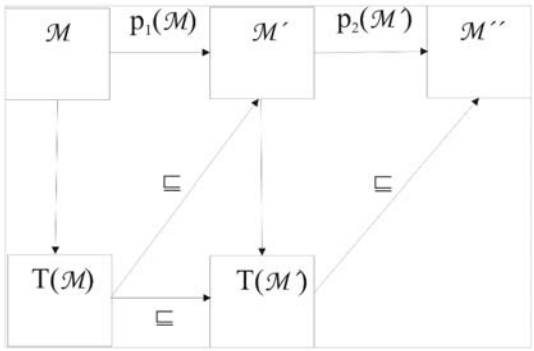


FIGURE 4. Transforming a Platform Independent Models M into a Platform Specific Model M"

Validation of the PSM is performed within the Event B framework, where we can show that the PSM M' is a refinement of the transformed platform independent model T(M). To ensure that the requirements (1)-(4) given above are satisfied we introduce a number of additional proof obligations in Event B. These proof obligations concerns deadlock freeness of the state-machines and enabledness of the transitions in the PSM that are refined from transitions in the PIM.

The transformation method is described in more detail in a technical report [BNOW06]. As future work, we aim at to applying the method on a case study to investigate its practicality and to develop reusable patterns. The method is not limited to only the small subset of UML state-machines given in the paper, but it can be extended to consider more features from the UML standard. UML is well known in industry and therefore this type of transformation rules can be beneficial in many application areas.

4.2.3 Model Checking and Animation

The use of the ProB tool has been extensive in the NoTA project. There have been two uses in particular:

- Testing through animation
- Demonstrations

In the first case ProB has been used primarily as an animator and thus a way of "executing" the B specifications. The method employed generally has been to write a piece of specification, ensure its correctness in terms of typing and consistency using the AtelierB theorem prover and then giving that verified specification to ProB.

There are three ways of working with ProB that we have used:

- Random animation
- Model checking

- Directed animation

In the random animation case we have either used ProB's random mode and allowed it to randomly apply (usually about a 1000 or so) operations or manually selected operations to check a particular piece of specification.

The main results here have been that we can be sure that the specification we have written does what we think it will do. It is the case with verification that incorrect specifications in terms of "customer correctness" are often passed. One area that has been of great benefit is checking whether the correct function type has been applied: total vs. partial, function vs. injection vs. surjection vs. bijection.

The model checking mode has been used to extensively check the specification in much the same way as the random mode. Here the results have been that particular cases which usually correspond to unproven obligations in the specification have been definitively identified.

Directed animation has been made from the test cases/use cases that were suggested or given during the domain modelling or requirements stages. These we have encoded in CSP and used ProB to run these cases. Initial cases were of course simple, for example the CSP specification of the interconnect node start up and shutdown was simply:

```
MAIN = initialise -> notify_resource_manager_location ->
shutdown;;
```

Over time more cases were encoded like this, either from a specific use case or from a manual animation run. If more than a few similar runs were made then we would often explicitly write the run as a CSP expression.

Finally after amassing a collection of CSP expressions it was possible to write a single expression that corresponds to the expected behaviour of the system:

```
MAIN = initialise -> notify_resource_manager_location ->
        (SERVICELIFE ||| SOCKETMANAGEMENT);;

SERVICELIFE =
  register_with_ResourceManager?NN -> register_with_ICNode?SS ->
  ( (SERVICE(SS) ||| SEARCH(NN)) ; FINISH_SERVICE(SS,NN));;

SEARCH(NN) = search!NN -> SEARCH(NN);;
SEARCH(NN) = skip;;

SERVICE(SS) = activate!SS -> (CONNECTIONS ; (deactivate!SS ->
SERVICE(SS)));;
SERVICE(SS) = skip;;

FINISH_SERVICE(SS,NN) = deregister_with_ICNode!SS ->
    deregister_with_ResourceManager!NN -> SERVICELIFE;;

CONNECTIONS = connect -> CONNECT_RUN;;

CONNECT_RUN = send -> CONNECT_RUN;;
CONNECT_RUN = get -> CONNECT_RUN;;
CONNECT_RUN = local_disconnect -> skip;;
CONNECT_RUN = remote_disconnect -> skip;;
CONNECT_RUN = local_close -> skip;;

SOCKETMANAGEMENT =
      automatic_close -> SOCKETMANAGEMENT []
      socket_send_process -> SOCKETMANAGEMENT []
```

```
socket_receive_process -> SOCKETMANAGEMENT;;
```

These CSP traces could either be run manually with ProB explicitly restricting the choices of operations to be run next or automatically as a model checker over the trace admitted by the given CSP expression.

This form of model based testing was done manually as there is no automatic way - yet - of sending various sets of traces to ProB.

The second usage of ProB was as a demonstration engine. Presenting the work here using the theorem prover and demonstrating correctness through proof is almost impossible for anyone to understand. However, hands-on demonstrations given to developers and management proved successful especially when they themselves could drive the animation.

This we found has one great advantage over a prototype in that the customer does not get confused or sidetracked by a user-interface mock-up and concentrates on what the system is supposed to be doing and thus it focusses the customer on what they have precisely asked for.

4.2.4 Model-based Testing in NoTA

We describe here the testing framework for the NoTA interconnect. We describe two testing approaches and explain the chosen test configuration. The aim of this document is explain the construction of the tester model and tester system at technical level without showing the actual code. The target for testing described in this document is the H_IN part of the NoTA system, but the approach is general. In particular the same approach can be applied for testing services that use NoTA primitives.

s described earlier the crux of Model Based Testing is to model the behavior of the System Under Test (or Behavior Model) and then let a tool perform testing by analyzing this tester model and sending messages over to the SUT. The analysis is redone based on the responses from the SUT, thus the testing is on-the-fly. It can be thought that the tester tries to execute the tester model concurrently in sync with the SUT. A response from the SUT that is not valid in the tester model is considered to be an error and is reported as such.

A point worth emphasizing is that the tester model (or behavior model) is not constructed from the external tester point of view, but from the system point of view. The tester point of view is the one that is usually taken when writing test cases, for instance: press button labeled â1â (send â1â to SUT) and observe that character â1â gets written on the display (receive â1â from SUT display). The system point of view describes what the system does â the previous example would then look like: wait for a key press, when key is pressed (receive from outside) and the key is â1â then emit character â1â on the display (send to outside via display). So we are modeling the behavior of the system, not the behavior of the test case.

A way of visualizing this is that the tester tool tries to ‘invert’ the tester model: if the tester model has a ‘receive’ then the tester tool must send a corresponding message to the SUT and if the tester model has a ‘send’ then the tester must expect a message from the SUT. As the

tester model describes the H_IN network the tester tool emulates the possible messages that the H_IN nodes send.

Another way of thinking is that the tester model is a reference implementation for the SUT. Thus the tester model that is being created is a design level artifact.

There are essentially three kinds of models: the tester model, the observer models and the abstract model.

The most important model is the already mentioned tester model. It describes the behavior of the tested system at an accuracy that allows it to be connected to the tested system. More accurately, the model connects to the tested system via some interfaces that allow communicating with the tested black box. Note that these interfaces may be wrappers on top of some more detailed interface.

When testing is performed based on the tester model the whole behavior described in the tester model is tested against the SUT and all parts of the tester model behavior as good as any other part. From the tester model point of view it makes no sense to say that a certain feature (a sub-behavior of the model) is being tested at some point. When some criteria, like testing time or structural criteria over the model for instance, are met and no errors have been found it can be said that the behavior described in the model has been tested. It is possible to look at the traces produced by the testing and then deduce that what was happening at given time â but from model point of view this is still irrelevant, there are no distinguished features.

However, the complete behavior of the model is very large so that it is not feasible to enumerate them all and in the model there still are behaviors that are very relevant. In case of H_IN its purpose is to transport data, so it makes sense to require that the testing exercises that part of the model in a way that a proof can be offered for completing that part of testing.

There are four ways of doing this: one is to observe the tested traces after the testing, second is to construct the behavior of the model so that relevant functionality is exercised, third is to tag the model so that it can be observed when certain tagged feature is tested and fourthly construct another model of the desired features.

These are related to the notion of coverage and they are explained in more detail in section 2.3.

If the fourth of the above possibilities is chosen, then it means that there is another model artifact: the observer model or observer automaton. These observers can be used in two roles over the test model: to guide the testing but also to observe the traces produced by testing afterwards.

There is also a third role for observers in ensuring a relation between two models, explained below.

A third kind of model exists, it is an abstract model of the system that is being tested. In this case this model is expressed in a language that concentrates on modeling the concepts of the system and allows then checking the consistency of the abstract model. Also it is possible to define features in a format that allows exhaustive checking of them against the abstract model. These features are similar to observer automatons. Due to complexity and size, the same analysis are not available for the tester model.

In order to use the results from the abstract model side on the tester model side, there are two approaches, which are closely related: use the features checked on the abstract model side as observers on the test model side and alternatively use the observer automatons to observe the traces produced by the abstract model as well to observe the traces from the tester model.

Note however that the power of the model-driven testing relies on letting the tool generate various permutations, so relying too excessively on guiding the testing, power of the approach can be diminished.

As the tester model itself is a reference implementation, there must be a way of ensuring the correctness of it. There exists a more abstract model of H_IN which has been written in description language B. Abstract models written in B can be automatically checked for consistency by means of a theorem prover and validated by means of a model checker. These methods increase the trust on the abstract model being correct. Unfortunately, the same analysis mechanisms cannot be used directly on the tester model; this is because there is not defined a mapping between the Lisp tester the B abstract model in such a way that the property of refinement can be proven, this is described below in more detail.

There are basically two ways of connecting the abstract model and the more concrete tester model. First one is to generate the concrete tester model from the abstract model which means adding (architectural) information to the abstract model. Currently this approach is not chosen, even some preliminary work had been done. Main reason for not pursuing this is that creating a compiler that is able to allow intelligently injecting details is too time consuming and hard.

The second approach is comparing the results of the model analyses. Here the analysis of the tester model (in presence of a black-box implementation) produces a trace of events. The analysis of the abstract model produces complete or partial state space that can be thought of a graph that contains all possible traces within that state space. For this reason we pick comparison of traces as the mechanism for linking the two models together. The assumption here is that the abstract model nevertheless has enough common elements with the tester model so that they can reasonably be compared â in this case it is the events or message primitives that are exchanged between the H_IN and its users.

To compare traces we use observer automatons, which are state machines which have message primitives on the transitions. These automatons operate on a trace and if the events of a trace lead to an accepting state, then the observer has witnessed the property for which it was designed for.

Figure 2 shows an example observer automaton that encodes the following property: a connection that has been established will be ended by the connection initiator (Hclose) or the peer (Hdisconnect_ind). The automaton has the following features: the events are messages that may have parameters, the parameter values may either be ignored (signaled by _) or remembered under a variable name, sockid in this case. The value stored under variable name may be matched later on. There is also a default or empty transition that is taken if no other transition can fire, that is when the event of a trace cannot be matched by any of the transitions in the state. The final or accepting state is denoted by a doublecircle.

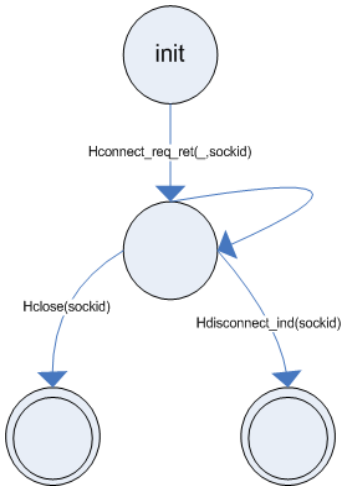


FIGURE 5. An Observer Automation

Note that the actual message representations may be different on the abstract model and on the tester model level, but the statemachine skeleton is still valid for both.

The B model checker can also use CSP descriptions to limit the validation run and a statemachine can be translated to CSP and vice versa. It may be that if the state space of the B model is very large the best course of action is to use the model checker in conjunction with the CSP properties to validate the property on the B side and then a corresponding observer automaton to validate the same behavior on the tester model side.

Note, that use cases that have been formalized using sequence diagrams can be transformed to CSP. This allows using the use cases to be used as properties to be checked by the B model checker. The H_IN implementation is generated by hand based on the B abstract model.

When the Lisp tester model tests the H_IN implementation and the automatons that are based on the CSP properties have been fulfilled we have demonstrated a refinement between the B model and the implementation.

With observer automatons the coverage issue on the tester model side is diminished, if the observer accepts, then a certain property has been observed. However, it may be that there is a need to include coverage like information for the automatons, for instance, how many times a certain property has been observed.

4.2.5 Hardware Description Language Generation

We have focussed our attention to the generation of hardware rather than the traditional focus of software. One problem that seems to repeatedly occur in software generation is targeting the Symbian platform with reasonable C++ code; this has lead to the adoption of strict manual processes in many cases as the construction of a code generator is too expensive in terms of time.

Hardware however provides a more beneficial target in that the gains through abstraction and correctness outweigh similar gains in the software arena due to the very explicit costs and difficulties incurred in hardware development.

Work has already been made regarding translating B to traditional hardware specification languages such as VHDL and SystemC. However the semantic gap between B and these languages is very large and the style of their usages very different. This makes code generation either impossible or at best convoluted with the result of inefficient and unworkable designs. The Bluespec language is based on an action system semantics more akin to that seen in EventB (and B). The relationship between the semantics of the languages is much closer to the point that the mappings are surprisingly easy. One must however take into consideration the hardware scheduling when making the mapping, but this can be taken care of through B's refinement mechanisms - we hypothesise automatically (but not describe here). As we can generate efficient hardware from Bluespec, this makes is more or less the ideal choice as a target for code generation.

Bluespec is a rule based declarative hardware specification language based on term rewriting. It is supported by a compiler capable of producing synthesisable SystemVerilog and a large set of libraries for common hardware IP blocks such as FIFOs, various register and memory structures etc.

The structure of a Bluespec description is in the form of an interface declaration and module declaration which implements an interface. All Bluespec methods must be declared in the interface and implemented in the module. Many modules may implement the same interface. We take the approach that a method is derived from an operation that is used to communicate with the environment and a rule for any operation which modifies the internal state.

One can see the similarities in the syntax of the languages in the two short code fragments shown below:

```
moveDown =
PRE
activity = DOWN &
currentPosition > requestedPosition &
currentPosition > 0
THEN
currentPosition := currentPosition - 1
END ;

request(ff) =
 PRE
ff : INTEGER &
 ff >= 0 & ff <= topFloor &
 not(ff = currentPosition) &
 availability = AVAILABLE
 THEN
 requestedPosition := ff ||
 availability := NOTAVAILABLE
 END

 av <-- is_available =
 BEGIN
 av := availability
END;
```

and in Bluespec:

```
rule moveDown(activity == Down &&
currentPosition > requestedPosition &&
currentPosition > 0 );
currentPosition <= currentPosition - 1;
 endrule
 method Action request(Int#(8) ff)
 if (availability == Available);
 if ( (ff >= 0) && (ff<=topFloor) &&
 (ff!=currentPosition) &&
 (availability == Available) )
 begin
 requestedPosition <= ff;
 availability <= NotAvailable ;
 end
 endmethod

 method LiftStatus is_available;
 return availability;
endmethod
```

Both B and Bluespec are based upon action system semantics which are an extension of Dijkstra's guarded commands [6]. B uses atomic, interleaving actions; Bluespec is similar but with clock-scheduling and rule priority. Bluespec's term rewriting aspect is similar to that of B and is adequate for many functional correctness properties.

Clock scheduling of rule introduces timing properties and picks a deterministic route through the transitions described by the term rewriting. This guarantees that the system always reaches the states predicted by the term rewriting rules but may never visit some states allowed by this.

These can be summarised such that in B, one picks any enabled action and executes its body (this guarantees atomicity), while in Bluespec one executes any enabled rules atomically. This is further re ned under scheduling: wait for a given clock event such as a rising clock edge and then execute the enabled set of rules. In practise a program in Bluespec will only 'visit' a subset of the states that a B specification can due to Bluespec's scheduler picking certain rules to re rather than potentially any at random. Of course there are pragmatic conditions which again limit this such as the amount of hardware logic that can be reasonably fitted into a clock period, resource limitations such as the readability of registers and energy consumption.

Any given speci specification in B can hypothetically be mapped to Bluespec but the scheduling policy and variable conflict rules will require additional guarding of the rules and methods in the Bluespec. There are three possibilities for re ning the B specification to construct a correctly functioning Bluespec program:

- Change of rule priorities
- Splitting of rules
- Constraining rules

Of these, the first is a modification to the architecting process that generates the Bluespec, the others can be made inside the B specification itself.

Often just changing the rule priorities inside the Bluespec has an effect. Bluespec calculates the rule priorities from the syntactical ordering of the rules in Bluespec program. If these rules can be reordered then sometimes the conflicts are removed.

Splitting the rules in the specification is made to leverage more parallelism in the Bluespec. The sometimes has the effect of making the scheduling easier such that one rule may be split such that the preconditions are much simpler and are more focussed to the actual actions inside the rule - that is there is less potential conflict between the precondition and the mechanics of the actions themselves. When using this 'pattern' it often becomes clearer which parts of a rule are causing conflicts and thus isolating scheduling problems is much easier.

Further constraining the rules by adding additional precondition statements basically restricts the firing of the rules such that the scheduling is more deterministic and fixed. This may also be accomplished by the addition of additional variables which effectively encode the behaviour of the scheduler.

When these changes are made to the B specification we can under some circumstances use the refinement mechanism in B directly to ensure that the modified B specification still adheres to the desired behaviour encoded in the most 'abstract' B machine. However, perculiarities in B and the semantics of refinement, especially in the latter cases of splitting of rules and encoding the scheduler through additional variables may cause problems with abstract forms of the additional rules and variables being required to be declared in the most abstract versions of the specification. Typically one might have to introduce operations of the form:

```
op = BEGIN skip END
```

to the abstract specifications and variables declared but without use inside the specification. These have no real effect other than to clutter the specification somewhat. The EventB language reduces these kinds of restrictions we seen in B; this will be investigates more later as at this point we have concentrated on the B language rather than EventB.

Given a Bluespec program generated from a B specification we can use existing synthesis tools to produce RTL and netlists for synthesis. Because we are working from a verified description this greatly reduces the development and testing times such that any increase in chip/area size and power consumption is annulled by the reduction in development, testing and debugging times. In the latter cases we are currently seeing approximately between -5% and +10% differences in power consumption and floor area compared to traditional VHDL based development for our current test designs. In the example the B and Bluespec are 100 lines each, the System Verilog RTL was 287 and this generated for a Xilinx Spartan FPGA: 48 FMAPS (24%), 5 HMAPS (5%), totalling 24 CLBs (24%) or for an Altera Cyclone FPGA: 23 I/O ATOMs, 54 LUTs (1%), 62 ATOMs (2%) for logic resources, 234 inputs on ATOMs and 10769 nets.

4.2.6 H_IN v2 Specification

The NoTA (Network on Terminal Architecture) Platform is begin designed to allow services in the service-oriented sense to run on a mobile device instead of the current component or modular based architectures. NoTA has been architected into four logical parts (see figure 7), one of which is the session/connection layer (ISO parlance) that provides the communication medium between the services. This layer is called the High Interconnect or HIN for short. We have concentrated on this particular layer and the interplay between UML and B for specification. The reason for the use of B was that the tools when we started this project were B based (AtelierB, ProB, U2B). Now that the EventB toolset is now available we are currently porting the specifications to EventB.

In this chapter we describe the development flow and the use of B in the specification of the HIN layer. In particular in the context of this document we concentrate on the following:

- The NoTA Domain Model
- The NoTA Architecture
- The HIN Layer
- The Interface partitioning

The domain model is effectively the structure of the concepts of a given system. In the case of NoTA we have gathered all the known concepts and constructed a domain model which describes the relationships between those concepts. The domain model is used to refine the definitions such that each concept becomes clean and consistent throughout the system as a whole.

At this stage of modelling we chose a very pragmatic formal way of modelling and only really concentrated on the invariants which were immediately obviously. This gave us a much quicker start and more time concentrating on developing the specification rather than concentrating on very small parts precisely. This style was much more applicable and amenable to those developers which were either accustomed to more implementation oriented styles or so called agile methods.

Figure 6 shows the current domain model of NoTA written in a subset of the UML modelling language. It is important to note that the domain model is independent of any implementation platform and language. We also use the domain model in the object oriented sense which extends the entity-relationship's structural ideas to encompass behaviour as well.

Note that the domain model contains both graphical and textual representations; we argument the graphical model with constraints written in OCL. This is used because a graphical model can not express certain constraints.

We start with the traditional focus of a NoTA system which is the Interconnect Node; this encompasses the behaviour and necessary structures to enable the NoTA system to function. In particular it manages the communication structures inherent in the NoTA system and also manages the communications with the resource management of the NoTA system. This

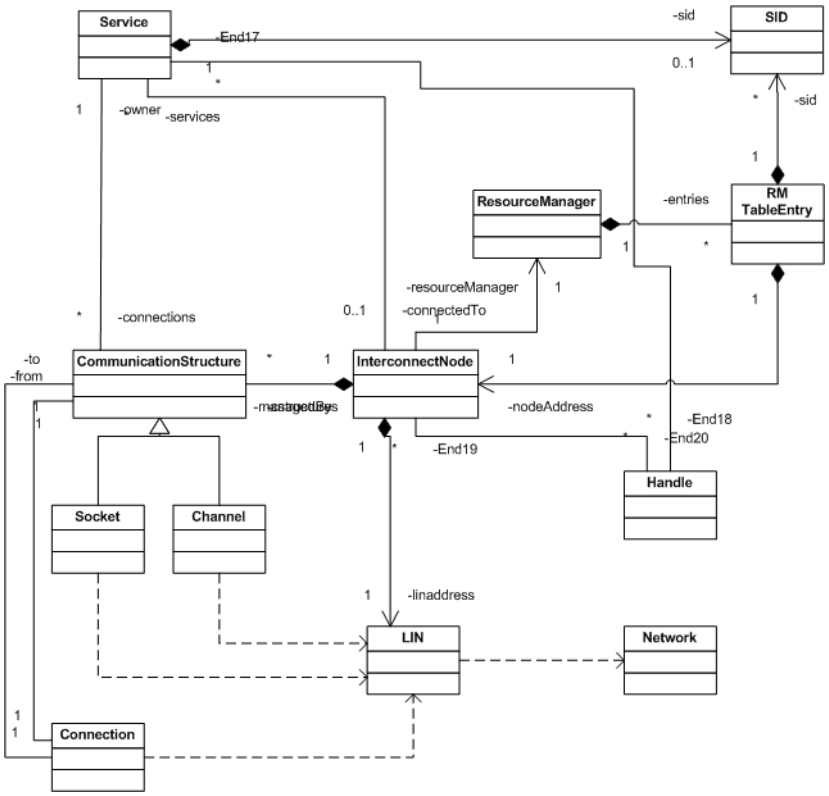


FIGURE 6. NoTA Domain Model

latter choice is partly and architectural decision but reflects the understanding that the resource manager is not a service-like element but an integral part of the NoTA framework.

Each interconnect node has an address obtained from a lower level addressing mechanism - in NoTA this is generally known as the LIN layer. Each interconnect node has a unique address:

```
context InterconnectNode
    self.allInstances->forAll(i1,i2 |
       i1.linaddress=i2.linaddress implies i1=i2 )
```

The resource manager itself is an element that manages the information about services as reflected in the structure of the resource manager table entries (RMTableEntry) that it contains. This particular concept is constructed out of a service ID (SID) and information about to which interconnect node that service ID is connected.

Given the two above concepts we can write the constraints relating these together. The first invariant states the available interconnect nodes are always a subset of the interconnect nodes addressed in the resource manager tables. This ensures that the resource manager can never have a reference to an interconnect node that does not exist.

```
context ResourceManager
    entries.nodeaddress->includes (InterconnectNode)}
```

The resource manager tables use the SID value as the key and thus this must be unique:

```
context RMTableEntry
```

```
 oclallInstances->forAll(r1,r2 |
   (r1.sid = r2.sid) implies(r1 = r2) )}
```

The service concept is also one of the most central to NoTA. This embodies the actual entity that for the user appears to perform all the work. As far as NoTA is concerned all services are identical and just pass data between themselves using the available communication structures. Each service may have a service ID (SID) - if none is present then the service has not registered with the system as is thus unknown to other services. However all services must connect to an interconnect node.

The known services to the resource manager must be the same as known to all the interconnect nodes in the system:

```
context InterconnnectNode
  services.sid = resourceManager.entries.sid
```

And similarly to interconnect node addresses, service SID's must be unique to each registered service:

```
contex Service
 allInstances->forAll(s1,s2 |
    (s1.sid = s2.sid)\oclimplies (s1 = s2) )}
```

Note that when the above invariant is combined with the sid uniqueness invariant in the resource manager table then these guarantee that the SID is unique across the system as a whole.

The next concept is that of the communication structure which can be further specialised into sockets and channels. Each basically behave in a similar manner as captured by the generalisation/specialisation relationship. The communication structures are managed by an interconnect node but are effectively owned by a given service. For security reasons only services that are connected to a given interconnect node can use sockets issued by that interconnect node:

```
context InterconnectNode
   services.connections = cstructures
```

The communication structures utilise a connection between themselves. This connection depends upon a lower layer concept known as LIN. A communication structure can not connect to itself:

```
context Connection
 to != from
```

and sockets and channels can not be connected together

```
context Connection
 to.OclType = from.OclTyp}
```

The Handle concept is not fully defined in this domain model at this time - this reflects the current usage and option about this structure inside NoTA.

The LIN and Network concepts are provided for completeness and not further developed at this point. As we shall see these become more architectural in nature rather than conceptual.

The NoTA Architecture is based around four layers and can be seen in figure 7. One can roughly align these with the OSI seven layer model with the Service layer providing application and presentation functionalities, the HIN providing session and transport while the LIN and Network layers are concerned with the OSI network, data and physical functionalities.

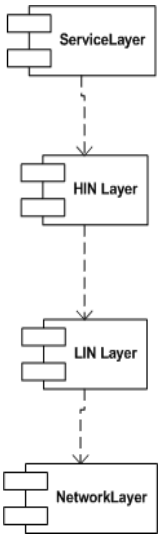


FIGURE 7. NoTA Architecture

These four layers are used to partition the NoTA domain model such that we can make a clear separation about what is specifically related to each layer. From this we can also identify and partition the interfaces between those layers.

The domain model given in figure 6 was mapped such that the Service, LIN and Network concepts were mapped to their related parts in the architecture model almost directly. This is as we had expected given the information about the system anyway - initial models of NoTA concentrated more on the so called "High Interconnect" layer anyway.

The result is that we had a system that is based around two concepts in this layer: the interconnect node itself and the resource manager. There were two data types present: SID and Handle - the latter not being well defined at this time.

We also made the architectural choice that the communication structure would be some kind of data structure within an interconnect node as the composition relationship in the domain model suggested. Further architecting or design of the resource manager was made elsewhere.

Once the domain modelling had been made the interconnect node's specification was to continue using the B modelling language rather than "UML". The B specification was arranged around the Interconnect node with addition B machines used to "simulate" the resource manager, the common data types present in the system and the behaviour of the handle objects. A previous experiment has been made applying U2B to the domain model but the B produced was too complex and there were issues regarding the locality of some of the operations. By choosing to work from after the system had been architected and then just on the design of a single interconnect node simplified the specification.

The OCL constraints seen in the domain model were mapped by hand into equivalent B expressions in the invariant. Socket and channel structures were mapped to function expressions in the state variables.

Most of the work regarding the behaviour of the interconnect node and its related structures was made at this level rather than at the domain level using OCL. B has tool support which is not present in a mature enough form with OCL.

Much of the design work later focussed on the behaviour of the sockets and channels and the peculiar issues related with the teat-down scenario where a socket is closed by one of its users

(the initiator or initiatee services) when there is still data either waiting to be sent or gotten from the input buffer of the socket. NoTA tries not to lose data so a socket shutdown might only be partial until both parties agree to close the socket. The design of the buffered socket is presented below in B and also pictured using a state machine in figure 8.

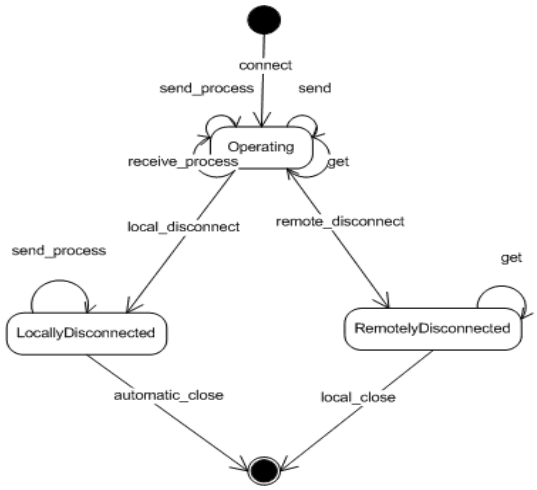


FIGURE 8. Socket State Machine

The socket structure invariant:

```
 issued_sockets <: SOCKET &
  local_sockets : issued_sockets --> dom(local_services) &
  socket_send_buffer : dom(local_sockets) --> INTEGER &
  socket_receive_buffer : dom(local_sockets) --> INTEGER &
  dom(socket_send_buffer) = dom (local_sockets) &
  dom(socket_receive_buffer) = dom (local_sockets) &
  socket_state : dom(local_sockets) --> SOCKET_STATE &
  dom(socket_state) = dom(local_sockets) &
  issued_sockets = dom(local_sockets) &
  !ss . ( ss : dom(socket_send_buffer) =>
                socket_send_buffer(ss) <= send_buffer_maxsize ) &
  !ss . ( ss : dom(socket_receive_buffer) =>
                  socket_receive_buffer(ss) <=
receive_buffer_maxsize ) &
  !ss . ( ss : dom(socket_send_buffer) =>
                  socket_send_buffer(ss) >= 0 ) &
  !ss . ( ss : dom(socket_receive_buffer) =>
                  socket_receive_buffer(ss) >= 0 ) &
  send_buffer_maxsize > 0 &
  receive_buffer_maxsize > 0 &
```

The socket tear-down operations:

```
err <-- socket_local_disconnect(sid,soc) =
   PRE
     icnode_state = RUNNING &
     sid : dom(local_services) &
     soc : issued_sockets &
     local_sockets(soc) = sid &
     socket_state(soc) = OPERATING
   THEN
     CHOICE
       err := SOCKET_DISCONNECT_ERROR
     OR
       err := SOCKET_DISCONNECT_OK ||
```

```
        socket_state(soc) :=  LOCALLY_DISCONNECTED
      END
   END ;

socket_remote_disconnect(soc) =
   PRE
       icnode_state = RUNNING &
       soc : issued_sockets &
       socket_state(soc) = OPERATING
   THEN
      socket_state(soc) := REMOTELY_DISCONNECTED
   END ;
```

The service-socket operations for the sending and obtaining of data from the socket buffers:

```
socket_send(sid,soc,data) =
   /* data parameter here for completeness */
   PRE
    icnode_state = RUNNING &
    sid : dom(local_services) &
    soc : dom(local_sockets) &
    local_sockets(soc) = sid &
    socket_state(soc) = OPERATING &
    socket_send_buffer(soc) < send_buffer_maxsize &
    data : DATA
   THEN
    socket_send_buffer(soc) :=  socket_send_buffer(soc) + 1
   END;

   dd <-- socket_get(sid,soc) =
   /* data parameter here for completeness */
   PRE
    icnode_state = RUNNING &
    sid : dom(local_services) &
    soc : issued_sockets &
    local_sockets(soc) = sid  &
    ((socket_state(soc) = OPERATING) or
(socket_state(soc)=REMOTELY_DISCONNECTED)  )&
    socket_receive_buffer(soc) > 0
   THEN
    socket_receive_buffer(soc) := socket_receive_buffer(soc) - 1
||
    dd :: DATA
   END ;
```

Channels work in an analogous way with some minor differences actually occurring inthe way data is send and received; also channels are uni-directional.

The fault tolerance pattern work by Bostrom et al is being applied to the above specification of sockets to assist in dealing with certain kinds of synchronisation problems.

From the specification at present, work has been progressing on mapping this into C and C++ for implementation on the Linux and Symbian platforms. Once this work is complete, it will be then be tested as described earlier. We will also be able to compare the code generated/developed via the formal specification with “agile developed” code from the previous version of this software.

4.3 Demonstrators

- The demonstrators for this case study will include the following:
- The domain model and architecture descriptions

- The outline of the development flow from UML to B/EventB and its place within the "model-based" development paradigm
- The B/EventB specification
- A demonstration of test cases in natural language, CSP and Comformiq Lisp
- A prototype of applying fault tolerance patterns to the B/EventB specification
- A prototype of the B to Bluespec code generator

In the previous sections we have outlined the development that has taken place in order to demonstrate the above.

4.4 Future Work

Up until now the work in this case study has focussed on the development of the specification within a live project and ensuring that the use of formal methods and the associated tools fits in with existing methods of working and are also accepted by practitioners who are not used to these methods and languages.

We have results from this work that show that the impact of using formal methods and the tools within the Rodin project do not affect the development structures already in place to any great degree. There does admittedly have to be some training and the use of coaching here has proven to be of great value. The major result however has been the speed at which a specification has been and can be developed and the confidence by the practitioners that the work is correct in both the verification and validation aspects. We can show approximately a 25%-35% increase in productivity over a one year period - the precise figures are not available at this time due to confidentiality reasons but will be published in due course.

With regards to the Rodin toolset we have taken the approach to be very hands-off in order for the tool set to develop to a point where it can be successfully employed by the practitioners who will then suggest improvements and so on. This mirrors often how tools are employed in that vendors normally release a tool for a generic market and customise as necessary later.

One addition reason for being hands-off is that formal methods are treated with some suspicion in industry and providing a tool which immediately requires a lot of training and customisation would only serve to reinforce that ideas. However it can be taken as a compliment to the developers of the various Rodin tools - the platform, U2B, ProB - that their acceptance within our development group is rather high even with the early version we see here; of course with in the Rodin project there is quite a degree of interaction anyway regarding the development of the tools.

During the 3rd year of the Rodin project we will concentrate more on integrating all the pieces we have presented here together to produce a more consistent whole.

Also during the 3rd year the NoTA project will effectively end and be replaced with three continuation projects in Nokia which further develop the ideas inside NoTA in both the academic/research area and also as a product.

4.5 References

[4.1]Kim Sandstrom, Ian Oliver (2006). A UML Profile for Asynchronous Hardware Design. In Proceedings: SAMOS IV Workshop, Greece.

[4.2]P. Bostrom, M. Neovius, I. Oliver and M. WaldÃ©n. Formal Transformation of Platform Independent Models Into Platform Specific Models in MDA. TUCS Technical report, 759, Turku, Finland, 2006

[4.3]I. Oliver. Model Based Testing and Refinement in MDA Based Development. In: Pierre Boulet (ed.) Advances in Design and Specification Languages for SoCs. The ChDL Series, Springer, 0-387-26149-4, 2005

[4.4]C. Snook and M. Walden. Refinement of Statemachines Using Hierarchical States, Choice Points and Joins. Presented at: REFINENET'05 - Refinement Workshop, University of Manchester, UK, October 2005,

[4.5]I Oliver. A Demonstration of Specifying and Synthesising Hardware using B and Bluespec. In: Proceedings of Forum on Design Languages FDL'06. Darmstadt, Germany

[4.6]K. Kronlof, S.Kontinen, I.Oliver, T.Eriksson. A Method for Mobile Terminal Platform Architecture Development. In: Proceedings of Forum on Design Languages FDL'06. Darmstadt, Germany

[4.7]I Oliver and V.Luukkala. On UML's Composite Structure Diagram. SAM06, May 31-June 2, Kaiserslautern, Germany

[4.8]I Oliver. UML and B in Industrial Development (Abstract). Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis 06191. May7-12 2006. Dagstuhl Schloss, Germany.

SECTION 5. CASE STUDY 4: CDIS AIR TRAFFIC CONTROL DISPLAY SYSTEM

5.1 Introduction

This section summarises progress during Year 2 on the CDIS case study. Contemporary tool support has been used to develop a formal specification of CDIS. CDIS is a computerised system that provides important airport and flight data for the duties of air traffic controllers based at the London Terminal Control Centre. Each user position is a workstation that includes a page selection device (to select CDIS pages) and an electronic display device (to display the selected pages). The original system was developed by Praxis[1] in 1992 and has been operational ever since. This system is an example of an industrial scale system that has been developed using formal methods. In particular, the functional requirements of the system were specified using VVSL [5.6] — a variant of VDM [5.5]. The formal development resulted in about 1200 pages of specification documents and about 3000 pages of design documents. The reliability of the delivered system is encouraging for formal methods in large scale system development because the defect rate was a considerable improvement on other similarly sized projects [5.7].

During the first year of the project a useful subset of the CDIS specification was defined, reviewed and distributed. Examples of problem areas in the original CDIS development were identified:

1. The lack of any formal proof in the original development.

2. The difficulty in comprehending the original specification and the difficulty of modularising the specification.

3. The difficult of dealing with distribution and atomicity refinement.

In Year 2 we focused on addressing items 1 and 2 above, though we also made some progress towards dealing with item 3. Two different attempts at reworking subset specification commenced: a “translation” approach a “specify equivalent system from scratch” approach. It was quickly found that the translation approach was not sensible and we focused in Year 2 on the second approach of specifying the system over again.

Redeveloping an existing system also allows us to reflect on the lessons learned from the original development. Our aim in this section is to demonstrate how we have attempted to overcome the lack of comprehensibility and formal proof of the original CDIS development by adopting a methodology that makes use of available tool support

[1] Praxis High Integrity Systems Ltd., U.K.

in an effective way. The major outcome in Year2 for CDIS was the elaboration of an approach for large scale formal development.

The work on CDIS in Year 2 was presented at a number of internal RODIN meetings:

- Zurich plenary meeting (September 2005)
- Newcastle CDIS meeting (February 2006)
- Aix-en-Provence plenary meeting (April 2006)

The CDIS work heavily influenced work on adding records to Event-B which is described in a paper presented at FM06 in Canada [5.2]. In addition, the approach for large scale formal development elaborated for CDIS in Year 2 is presented in a paper submitted to ISoLA 2006 [5.3].

5.2 Major directions in case study development

In order to keep the case study manageable in the context of the RODIN project, a subset of the original CDIS has been carefully chosen for redevelopment [5.8]. However, rather than focusing on individual aspects of CDIS, a 'vertical slice' has been taken so that all of the interesting features of the system are covered (albeit in a lesser form). At the heart of the CDIS subset is the 'core specification' that gives the functional properties of the system, and shall be the focus of this section. In addition to the core specification, there is a concurrency specification and a description of the user interface.

The Original Specification

The core specification is only one part of the overall CDIS documentation. It gives an idealised view of the entire functional behaviour of the system. (The design document states how this is actually realised.) In order to avoid ambiguity, in this section we will often refer to the core specification as 'the original VDM specification'.

The core specification consists of a number of VVSL modules, each of which contains type, constant and state definitions. (The bulk of the specification is made up of Boolean functions that are used in the pre/post conditions of other definitions.) A module can import other modules so that the imported definitions are available in the importing module. This gives a VVSL specification its structure. This approach encourages a bottom-up development in which the overall specification emerges from the way in which its modules are combined.

The core specification of CDIS comprises 15 modules. However, we can identify three main parts (or contexts):

- Airport-related data. This concerns airport-specific values such as weather or runway information. The $Meta\_data$ module identifies the airport attributes and their value types. Functions are defined to update the values of the attributes. The $Airport\_records$ module declares a state variable that holds all of the actual values of the attributes.

- Page-related data. This gives a device-independent and data-independent record of the pages that can be displayed by CDIS. Types are declared to model the layouts of pages. Actual pages are held in the state variables declared in the $Pages$ module.

- Display-related data. This concerns the physical devices that are used to retrieve and display information.

Other subsidiary modules such as the date/time module are concerned with other important features of CDIS. By far the largest module in the core specification is $EDD\_displays$ that contains the operations of the system. All of the modules listed above are imported by $EDD\_displays$ to enable the definition of the operations.

Conclusions Drawn

It is worth emphasising that the CDIS specification is necessarily complicated. Even though the core specification has been criticised for its complexity, it is unrealistic to expect any significant improvements in the size of a specification that captures all aspects of CDIS, regardless of the notation used. However, the bottom-up construction in VVSL forces a level of specification that is too detailed to get an appreciation of the overall system behaviour.

Too much complexity also precludes formal analysis. In order to reason about a specification formally, it is necessary to keep the level of detail as simple as possible. Otherwise mathematical proof becomes infeasible. Analysing monolithic specifications such as the CDIS core specification would be beyond the capabilities of contemporary formal methods tools without intense human intervention. This was not an issue during the original CDIS development because tool support was largely unavailable, and large-scale formal analysis was out of the question.

5.3 Progress in developing the demonstrators

In this section we describe our formal development of CDIS in Event B. In order to get a better overview of the entire system, we follow a top-down approach. At the top level, we ignore all of the airport-specific features to produce a specification describing a generic display system. Through an iterated refinement process, we introduce more

features into the specification until all of the CDIS functionality is specified. This procedure is supported by the tool B4Free. At each step the tool generates a number of proof obligations which must be discharged in order to show that the models are consistent with their invariants. Since each refinement introduces only a small part of the overall functionality, the number of proof obligations at each step is relatively small (approximately less than 20).

5.3.1 Abstract Specification

The purpose of CDIS is to enable the storage, maintenance and display of data at user positions. If we ignore specific details about what is stored and displayed then CDIS becomes a 'generic' display system. We begin by constructing a specification for a generic system (which will be, of course, somewhat influenced by the original VDM specification) and, through subsequent refinements, introduce more and more airport-specific details so that we produce a specification of the necessary complexity, and reason about it along the way. By providing a top-down sequence of refinements it is possible to select an appropriate level of abstraction to view the system: an abstract overview can be obtained from higher level specifications whilst specific details can be obtained from lower levels.

Meta Data Context.

Rather than specifying individual airport attributes (such as wind speed) as state variables of a particular value type, two abstract types are introduced that correspond to the collection of attribute identifiers and attribute values. This allows us to represent the storage of data more abstractly as a mapping from attribute identifiers to attribute values.

CONTEXT *META_DATA*
SETS *Attr_id* **;** *Attr_value*
END

Pages Context.

The pages of CDIS are device-independent representations of what can be displayed on a screen. Each page is associated with a page number, and each page consists of its contents.

CONTEXT *PAGE_CONTEXT*
SETS *Page_number* **;** *Page_contents*
END

Displays Context.

At this abstract level, we model the physical devices with which the users interact with the system. However, we only need to acknowledge that each position is uniquely identified (by its EDD_id), each user position has a type, and each user position has a physical display. Some user positions are 'editors' which have the capability of manipulating data and pages.

CONTEXT *DISPLAY_CONTEXT*
SETS *EDD_id* **;** *EDD_type* **;** *EDD_display*
CONSTANTS *EDDs , EDIT , EDITORS*
PROPERTIES
$EDIT \in EDD\_type \wedge$
$EDDs \in EDD\_id \rightarrow EDD\_type \wedge$
$EDITORS \subseteq EDD\_id \wedge$
$EDITORS = EDDs^{-1}[\{EDIT\}]$
END

Merge Context.

By merging the previous three contexts (via a **SEES** clause), we can declare a function that can determine the actual display, given the appropriate information. In declaring this function, we use an unfamiliar syntax. In [5.2], we have proposed the introduction of a record-like structure to Event-B. This proposal does not require any changes to the semantics of Event-B, but it gives us a succinct way to define structured data. The declaration of $Disp\_interface$ in the **SETS** clause of the following context is an example of our proposed syntax

CONTEXT *MERGE_CONTEXT*
SEES *META_DATA , DISPLAY_CONTEXT , PAGE_CONTEXT*
SETS *Disp_interface* :: *data* : *Attr_id* $\rightarrow$ *Attr_value*,
contents : *Page_contents*
CONSTANTS *disp_values*
PROPERTIES *disp_values* $\in$ *Disp_interface* $\rightarrow$ *EDD_display*

The type $Disp\_interface$ is a record comprising two fields $data$ (of type $Attr\_id \rightarrow Attr\_value$) and $contents$ (of type $Page\_contents$). This record type defines the interface to the function $disp\_values$. The intention is that, given a database of values and the device-independent representation of a display, $disp\_values$ calculates what is actually displayed (i.e. it returns a value of type $EDD\_display$). The benefit of using a record type is that it can be refined by adding extra fields (see [5.2] for more details). This is necessary because the actual display is dependent on parameters that are introduced during the refinement stages. The extension of record types through refinement allows us to modify the interface accordingly (an example of this is given in Section 5.3.3).

As in the original CDIS specification, the fact that we represent $disp\_values$ so abstractly does not undermine the value of the specification. The dynamic part of the specification (shown below) focuses on updating attributes and pages, and defines the pages selected at user positions.

5.3.2 The Abstract Model: A Generic Display

The variable $database$ represents the stored data, and $page\_selections$ records the page number currently selected at a user position. Note that this is a partial function which means that user positions are not obliged to display a page. The variable $pages$ is a partial function mapping page numbers and page contents. The variable $private\_pages$ holds the page contents of a page prior to release. This is intended to model an editor's ability to construct new pages before they are made public. Finally, trq models the 'timed release queue' that enables a new version of a page to be stored until a given time is reached, whereupon it is made public.

MACHINE *ABS_DISPLAY*
SEES
META_DATA, DISPLAY_CONTEXT, PAGE_CONTEXT, MERGE_CONTEXT
VARIABLES *database , pages , page_selections , private_pages , trq*
DEFINITIONS
inv $\widehat{=}$
$database : Attr\_id \rightarrow Attr\_value \wedge$
$pages : Page\_number \nrightarrow Page\_contents \wedge$
$page\_selections : EDD\_id \nrightarrow Page\_number \wedge$
$private\_pages : Page\_number \nrightarrow Page\_contents \wedge$
$trq : Page\_number \nrightarrow Page\_contents \wedge$
$\mathsf{ran}(page\_selections) \subseteq \mathsf{dom}(pages)$
INVARIANT *inv*
INITIALISATION *database , pages , page_selections , private_pages , trq* : (*inv*)

Note that, in addition to type information, the invariant insists that pages can be selected only if they have contents. We keep the model simple by initialising the system to be any state in which the invariant holds.

Almost all of the operations given below correspond to operations defined in the original VDM specification. One exception is the **VIEW_PAGE** operation that uses the $disp\_values$ function to output an actual display. This is a departure from the original VDM specification but, since outputs must be preserved during refinement, it forces us to ensure that the appearance of actual displays is preserved.

UPDATE_DATABASE models the automatic update of data via the stream of data coming from the airports (see [5.8]), and **SET_DATA_VALUE** models the manual update of values (by editors). **DISPLAY_PAGE** enables any user to select a new page to be displayed, and **DISMISS_PAGE** removes a page selection. **RELEASE_PAGE** makes a private page public, and **DELETE_PAGE** enables an editor to delete the contents of a page. In addition to the manual release of pages (via **RELEASE_PAGE**), pages can be released automatically at specific times. **RELEASE_PAGES_FROM_TRQ** models the timed release of pages. However, at this stage no notion of time exists in the specification. Therefore, this operation selects an arbitrary subset of the pages from trq to be released. This is refined when we introduce a notion of time (as shown in Section 5.3.3). The operations use common B operators such as function overriding $\lhd\!\!+$, domain subtraction $\lhd$, and range subtraction $\rhd$.

UPDATE_DATABASE (*ups*) $\widehat{=}$
  PRE
    $ups \in Attr\_id \nrightarrow Attr\_value$
  THEN
    $database := database \lhd\!\!+ ups$
  END ;

SET_DATA_VALUE (*ei* , *ai* , *av*) $\widehat{=}$
  PRE
    $ei \in EDD\_id \wedge$
    $ai \in Attr\_id \wedge av \in Attr\_value$
  THEN
    WHEN $ei \in EDITORS$ **THEN**
      $database (ai) := av$
    END
  END ;

DISPLAY_PAGE (*ei* , *no*) $\widehat{=}$
  PRE
    $ei \in EDD\_id \wedge no \in Page\_number$
  THEN
    WHEN $no \in \mathsf{dom} (pages)$ **THEN**
      $page\_selections (ei) := no$
    END
  END ;

DISMISS_PAGE (*ei*) $\widehat{=}$
  PRE $ei \in EDD\_id$ **THEN**
    WHEN
      $ei \in \mathsf{dom} (page\_selections)$
    THEN
      $page\_selections :=$
        $\{ ei \} \lhd page\_selections$
    END
  END ;

$ed \longleftarrow$ **VIEW_PAGE** (*ei*) $\widehat{=}$
  PRE $ei \in EDD\_id$ **THEN**
    ANY *di* **WHERE**
      $ei \in \mathsf{dom} (page\_selections) \wedge$
      $di \in Disp\_interface \wedge$
      $data (di) = database \wedge$
      $contents (di) =$
        $pages (page\_selections (ei))$
    THEN
      $ed := disp\_values (di)$
    END
  END
  RELEASE_PAGE (*no*) $\widehat{=}$
    PRE $no \in Page\_number$ **THEN**
      WHEN
        $no \in \mathsf{dom} (private\_pages)$
      THEN
        $pages (no) :=$
          $private\_pages (no) \parallel$
        $private\_pages :=$
          $\{ no \} \lhd private\_pages$
      END
    END ;

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
  ANY *SS* **WHERE**
    $SS \in$
      $Page\_number \nrightarrow Page\_contents \wedge$
    $SS \subseteq trq$
  THEN
    $pages := pages \lhd\!\!+ SS \parallel$
    $trq := trq - SS$
  END ;

DELETE_PAGE (*ei* , *no*) $\widehat{=}$
  PRE
    $ei \in EDD\_id \wedge$
    $no \in Page\_number$
  THEN
    WHEN $ei \in EDITORS$ **THEN**
      $pages := \{ no \} \lhd pages \ \|$
      $private\_pages := \{ no \} \lhd private\_pages \ \|$
      $trq := \{ no \} \lhd trq \ \|$
      $page\_selections := page\_selections \rhd \{ no \}$
    END
  END ;

5.3.3 Refinement

The abstract specification described in the previous section omitted many of the features that characterise CDIS. However, this made it possible to give a broad overview of the system, including its state variables and operations, within a few pages. Now we use this specification as a basis for refinement in which the omitted details are introduced. We introduce a notion of time so that we can add age information to attributes, and add creation and release times to pages.

Adding Time

In terms of the CDIS subset, there are two main reasons for adding time: each piece of airport data has an age which affects how it is displayed, and the version of each page that is displayed is also time-dependent. In this refinement we shall once again use our proposed syntax for record types [5.2].

Time Context.

We begin by introducing a new context to the development. The set $Date\_time$ represents all of the different points in time. We also include a total ordering relation (leq) between these points.

CONTEXT *TIME*
SETS *Date_time*
CONSTANTS *leq*
PROPERTIES
$leq \in Date\_time \leftrightarrow Date\_time \wedge$
$\forall (a).(a : Date\_time \Rightarrow (a, a) : leq) \wedge$
$\forall (a, b).(a : Date\_time \wedge b : Date\_time \Rightarrow$
$((a, b) : leq \wedge (b, a) : leq \Rightarrow a = b) \wedge$
$((a, b) : leq \vee (b, a) : leq)) \wedge$
$\forall (a, b, c).(a : Date\_time \wedge b : Date\_time \wedge c : Date\_time \Rightarrow$
$((a, b) : leq \wedge (b, c) : leq \Rightarrow (a, c) : leq))$
END

Meta Data Context.

In order to record the age of a piece of data as well as its value, we refine the $META\_DATA$ context by defining a record type $Attrs$ with two fields $value$ and $last\_update$.

CONTEXT *META_DATA1*
SEES *META_DATA* , *TIME*
SETS *Attrs* :: *value* : *Attr_value*,
last_update : *Date_time*
END

Note that the range of $value$ is of our original value type $Attr\_value$. The gluing invariant of the refined model will ensure that the values of the entries in the refined database will match the corresponding entries in the original. The field $last\_update$ (of type $Date\_time$) records the time at which the value of the attribute was last updated.

This technique of 'wrapping' an abstract type in a refined type is a pattern that occurs frequently in our approach. In general, if $f \in I \rightarrow A$ is an abstract collection formed from abstract type A and in a refinement we wrap A in a record $B :: a : A, \cdots$, then abstract variable f is replaced by $g \in I \rightarrow B$ with gluing invariant $f = g; a$.

Pages Context.

We proceed by refining the pages context in a similar manner. We declare a record type $Page$ with two fields: $page\_contents$ holds the structure of a page, and $creation\_date$ holds the time at which a page was created. Note that this has nothing to do with the time at which the page is released. In order to model the timed release queue faithfully, we must associate a release date with every page on the queue. By using our

proposed syntax for record refinement [5.2], this is achieved by defining a subtype of $Page$ (called Rel_page) whose elements have an additional field called $release\_date$.

CONTEXT *PAGE_CONTEXT1*
SEES *TIME* , *PAGE_CONTEXT*
SETS
Page :: *page_contents* : *Page_contents*,
creation_date : *Date_time* **;**
Rel_page SUBTYPES *Page* WITH *release_date* : *Date_time*
END

Only pages of type Rel_page occur on the timed release queue. We shall see how the refinement of the operation **RELEASE_PAGES_FROM_TRQ** uses this additional information.

Merge Context.

Now that we have introduced a notion of time, the display function $disp\_values$ can be augmented so that the ages of the data in the database is taken into account when they are displayed. We change the interface of the function by adding a new field to $Disp\_interface$ called $time$. The operator 'EXTEND' is similar to the 'SUBTYPES' operator, but it adds fields to *all* elements of the record type.

CONTEXT *MERGE_CONTEXT1*
SEES
META_DATA , *DISPLAY_CONTEXT* , *PAGE_CONTEXT* ,
TIME , *META_DATA1* , *PAGE_CONTEXT1* , *MERGE_CONTEXT*
SETS EXTEND *Disp_interface* WITH *time* : *Date_time*
END

Whenever the function $disp\_values$ is called, the current time can be passed as a parameter so that the ages of the relevant data can be determined. In CDIS, the colour of a value when displayed indicates its age (although this detail is not included at this level of abstraction).

The Refined Model: A Timed Display.

The state variables and the operations of $ABS\_DISPLAY$ are refined to incorporate the timed context. Four of the variables in the refinement replace those of the abstract model. The invariant gives the relationship between these concrete variables and

their abstract counterparts. For example, the abstract variable $database$ is refined by $timed\_database$, and they are related because the attribute values held in $database$ can be retrieved from the $value$ fields in $timed\_database$.

REFINEMENT *ABS_DISPLAY1*
REFINES
ABS_DISPLAY
SEES
META_DATA, DISPLAY_CONTEXT, *PAGE_CONTEXT*, *MERGE_CONTEXT*,
TIME , META_DATA1 , PAGE_CONTEXT1 , MERGE_CONTEXT1
VARIABLES
timed_database ,
page_selections ,
timed_pages ,
private_timed_pages ,
dated_trq ,
time_now
DEFINITIONS
inv1 $\widehat{=}$
$timed\_database \in Attr\_id \rightarrow Attrs \wedge$
$timed\_pages \in Page\_number \nrightarrow Page \wedge$
$private\_timed\_pages \in Page\_number \nrightarrow Page \wedge$
$dated\_trq \in Page\_number \nrightarrow Rel\_Page \wedge$
$time\_now \in Date\_time \wedge$
$database = (timed\_database ; value) \wedge$
$\mathsf{ran}\ (page\_selections) \subseteq \mathsf{dom}\ (timed\_pages) \wedge$
$pages = (timed\_pages ; page\_contents) \wedge$
$private\_pages = (private\_timed\_pages ; page\_contents) \wedge$
$trq = (dated\_trq ; page\_contents) \wedge$
$\forall n . (n \in \mathsf{dom}\ (timed\_pages) \Rightarrow$
$(creation\_date (timed\_pages (n)), time\_now) \in leq) \wedge$
$\forall n . (n \in \mathsf{dom}\ (private\_timed\_pages) \Rightarrow$
$(creation\_date (private\_timed\_pages (n)), time\_now) \in leq) \wedge$
$\forall n . (n \in \mathsf{dom}\ (dated\_trq) \Rightarrow$
$(creation\_date (dated\_trq (n)), time\_now) \in leq)$
INVARIANT *inv1*

Some of the operations affected by the refinement are shown below.

UPDATE_DATABASE (*ups*) $\widehat{=}$
PRE $ups \in Attr\_id \pfun Attr\_value$ **THEN**
ANY ff **WHERE**
$ff \in Attr\_id \pfun Attrs \land$
$\mathsf{dom}\ (\ ff\ ) = \mathsf{dom}\ (\ ups\ ) \land$
$(\ ff\ ;\ value\ ) = ups \land$
$(\ ff\ ;\ last\_update\ ) = \mathsf{dom}\ (\ ff\ ) \times \{\ time\_now\ \}$
THEN
$timed\_database := timed\_database \rhd\!+ ff$
END
END

The parameter to the **UPDATE_DATABASE** operation maintains its type, but the **ANY** clause is used to construct a new mapping from $Attr\_id$ to $Attrs$ all of whose $last\_update$ components are assigned to the current time (to reflect the time of the update). This mapping is used to overwrite the appropriate entities in the timed database. An interesting refinement occurs in the operation **RELEASE_PAGES_FROM_TRQ**. Rather than selecting an arbitrary subset of trq to release, $time\_now$ is used to select those elements whose release date is earlier than the current time. The released pages (held in $timed\_pages$) are updated accordingly.

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
LET SS **BE** $SS =$
$dated\_trq \rhd \{\ rp \ \mid\ rp \in Rel\_Page \land (\ release\_date\ (\ rp\ )\ ,\ time\_now) \in leq\ \}$
IN
$timed\_pages := timed\_pages \rhd\!+ SS \ \|$
$dated\_trq := dated\_trq - SS$
END

Next, we introduce a new operation, called **CLOCK** that increases the current time by some unspecified amount. This operation models the passing of time.

CLOCK $\widehat{=}$
ANY *time_next* **WHERE**
time_next $\in$ *Date_time* $\wedge$
(*time_now* , *time_next*) $\in$ *leq* $\wedge$
time_next $\neq$ *time_now*
THEN
time_now := *time_next*
END

5.3.4 Another Refinement: Highlighting Manual Interaction

Several other aspects can affect the way values are displayed. One requirement of CDIS is that any manually updated values should be highlighted when they are displayed. Hence, with each attribute value, we need to record whether it was updated manually. Once again, we use our notion of record refinement to achieve this. The Boolean value associated with the new field *manually_updated* indicates whether the attribute's latest recorded value (accessed via the *value* field) has been input manually. In this case, we extend the record type *Attrs* as follows:

$$\text{EXTEND } Attrs \text{ WITH } manually\_updated : BOOL$$

If left unaltered, the existing B operations **UPDATE_DATABASE** and **SET_DATA_VALUE** would update this field nondeterministically, but we can refine them to assign meaningful values. In this case, the appropriate refinements are:

UPDATE_DATABASE (*ups*) $\widehat{=}$
PRE *ups* $\in$ *Attr_id* $\nrightarrow$ *Attr_value* **THEN**
ANY *ff* **WHERE**
ff $\in$ *Attr_id* $\nrightarrow$ *Attrs* $\wedge$
dom (*ff*) = dom (*ups*) $\wedge$
(*ff* ; *value*) = *ups* $\wedge$
(*ff* ; *last_update*) = dom (*ff*) $\times$ { *time_now* } $\wedge$
(*ff* ; *manually_updated*) = dom (*ff*) $\times$ { *FALSE* }
THEN
timed_database := *timed_database* $\lhd\!\!+$ *ff*
END
END

SET_DATA_VALUE (*ei* , *ai* , *av*) $\widehat{=}$
  PRE *ei* $\in$ *EDD_id* $\wedge$ *ai* $\in$ *Attr_id* $\wedge$ *av* $\in$ *Attr_value* **THEN**
    WHEN *ei* $\in$ *EDITORS* **THEN**
      ANY *aa* **WHERE**
        aa $\in$ *Attrs* $\wedge$
        value (*aa*) $=$ *av* $\wedge$
        last_update (*aa*) $=$ *time_now* $\wedge$
        manually_updated (*aa*) $=$ *TRUE*
      THEN
        timed_database (*ai*) $:=$ *aa*
      END
    END
  END

Since the operation **UPDATE_DATABASE** models the automatic update of values, all *manually_updated* fields are set to *FALSE*; **SET_DATA_VALUE**, which models a manual update, sets the *manually_updated* field to *TRUE*. Proving consistency of this form of superposition refinement is completely automatic.

5.3.5 Introducing Concrete Values

The ultimate aim of the refinement process is to construct a specification in which constants and variables are associated with concrete values and operations are defined to maintain the state accordingly. As part of this process, we have to separate an abstract type into subtypes. In the case of CDIS, this technique is used to introduce concrete attribute identifiers and value types into the specification. For example, the original VDM specification defines $Attr\_value$ as a union type made up of value types such as $Wind\_direction$ and $Wind\_speed$. Although union types do not exist in B, we employ the separation technique to achieve the same goal. We define a new context in which $Wind\_direction$ and $Wind\_speed$ are defined subtypes of $Attr\_value$[2].

[2]Even though $Attr\_value$ is not a record type, deferred sets such as this can be viewed as 'fieldless records'. By subtyping deferred sets, we can incorporate structure.

CONTEXT *META_DATAn*
SEES *META_DATA* , *META_DATA1*, $\cdots$
SETS
Wind_speed SUBTYPES *Attr_value* WITH *speed* : 0..99 **;**
Wind_direction SUBTYPES *Attr_value* WITH *dir* : 0..359 **;**
$\vdots$
END

Note that in this example we have refined $Attr\_value$ in two different ways. This is a reasonable thing to do (as discussed in [5.2]). The subtype $Wind\_speed$ has a single field $speed$ which ranges from values 0 to 99. Similarly, $Wind\_direction$ has a single field dir which ranges from 0 to 359.

This is just one of the many refinements needed to introduce concrete types. A further refinement introduces *AV_WIND_SPEED*, *MIN_WIND_SPEED* and *MAX_WIND_SPEED* as concrete attribute identifiers (since they appear in the core specification). From these refinements, it is necessary to specialise the update operations to ensure that only values of the correct type update the database. Abstract operations can be refined into one or more concrete operations. Previously, **SET_DATA_VALUE** updated any attribute identifier with any attribute value. Now it must be refined to a collection of operations, each referring to specific attribute identifiers and attribute values.

5.3.6 Error Handling

With every operation that assigns a meaningful value to a concrete attribute identifier (such as **SET_WIND_SPEED_VALUE** above), we must also say what happens when an attempt is made to assign an out-of-range value. This situation gives us the opportunity to handle potential errors in the update of CDIS explicitly. We define additional operations to handle updates with such out-of-range values. This approach in Event-B corresponds to the built-in error handling capabilities of VVSL. As an example, consider the following operation fragment (which is another refinement of the **SET_DATA_VALUE** operation) that attempts to assign an out-of-range wind speed.

SET_WIND_SPEED_ERROR (*ei* , *ai* , *av*) $\widehat{=}$
PRE *ei* $\in$ *EDD_id* $\wedge$ *ai* $\in$ *Attr_id* $\wedge$ *av* $\in$ *Attr_value* **THEN**
WHERE
ei $\in$ *EDITORS* $\wedge$
ai $\in$ { *AV_WIND_SPEED, MIN_WIND_SPEED, MAX_WIND_SPEED* } $\wedge$
av $\notin$ $Wind\_speed$

THEN

⋮

This operation only considers values outside the subtype $Wind\_speed$. The body of the operation should handle this anomaly in an appropriate way (such as by ignoring the update and issuing an error message).

5.4 Overview and Future Work

This section represents a methodological contribution to the construction of large formal specifications. Our experience shows that incremental construction through iterative refinement makes it feasible to apply tool-based formal analysis to large specifications. This increases our confidence in the specification greatly and provides the basis for tool-based formal development of a design and implementation. We also believe that this approach makes a large formal specification more accessible and comprehensible both to those constructing the specification and to others.

A key factor in our success was the construction of good initial abstractions capturing the essentials of the system concerned. Such a skill is not easily transferable of course, but by providing good examples, such as the one here, we can help others understand how to construct good abstractions. Beside this, we have provided a number of concrete techniques which are transferable to the construction of other large formal specifications. In particular we made strong use of the developmental pattern of extending records to add additional information to information structures and to extend function signatures in in refinement steps. We identified and made use of a related pattern of wrapping abstract types within record structures in a refinement step, providing a standard pattern for a gluing invariant. We also made use of record subtyping and record extension to differentiate structures in refinements and to add attributes to abstract deferred sets. These techniques allow us to avoid unnecessary clutter at the more abstract levels. The techniques are easily supported by existing B provers and our experience is that the associated proof obligations are mostly automatically discharged.

A drawback of the original development is the lack of continuity from the specification to the design. In the *idealised* view of the core specification, updates are performed instantaneously at all user positions, whilst there is an inevitable delay in the actual system because the information must be distributed to the user positions. Hence, there is no natural refinement of the original specification (in the usual sense of the word) to the design. We are investigating more novel notions of refinement in order to find a suitable link between the two viewpoints. The approach involves the use

of a richer abstract model that reflects the delay between updating the central database and those updates appearing at the individual displays. The idea is then to promote the existing idealised development in the context of the richer abstraction. In year 3 we will elaborate this approach and apply it to the idealised development.

In Year 2 we mostly made use of the B4free prover and the Click'n'Prove prover interface. In Year 3 we plan to apply the RODIN platform to the case study as a means of validating the platform.

References

[5.1] J. -R. Abrial: *The B Book: Assigning Programs to Meanings*, Cambridge University Press (1996).

[5.2] N. Evans and M. Butler: *Proposal for Records in B*, accepted for publication, FM06. `http://eprints.ecs.soton.ac.uk/12024/`

[5.3] N. Evans and M. Butler: *Incremental Construction of Large Specifications: Case Study and Techniques*, submitted to ISoLA 2006. `http://eprints.ecs.soton.ac.uk/12734/`

[5.4] A. Hall:*Using Formal Methods to Develop an ATC Information System*, Software, Vol. 13, No. 2, IEEE, March 1996.

[5.5] C. Jones: *Systematic Software Development using VDM*, Prentice Hall, 1990.

[5.6] C. A. Middleburg: *VVSL: A Language for Structured VDM Specifications*, Formal Aspects of Computing, Vol. 1, No. 1, Springer, 1989.

[5.7] S. Pfleeger and L Hatton: *Investigating the Influence of Formal Methods*, Computer, Vol. 30, No. 2, IEEE, February 1997.

[5.8] RODIN Deliverable D4: *Tracable Requirements Document for Case Studies*, http://rodin.cs.ncl.ac.uk/deliverables/D4.pdf, 2005.

SECTION 6. CASE STUDY 5: AMBIENT CAMPUS – THE LECTURE SCENARIO

6.1. Introduction

The main characteristics distinguishing the Ambient Campus case study from other Rodin case studies is the openness of the Ambient Intelligence (AmI) systems, the inherent autonomy of their components (agents), the asynchrony and anonymity of the agent communication, and the specific types of faults they need to be resilient to. To address these issues, we are specifically working on (i) ensuring interoperability of the independently developed agents, supporting this by original top-down stepwise system development methods, (ii) formally defining exception handling and structuring mechanisms suitable for this domain, (iii) proposing new modelling techniques capturing mobility and openness of the AmI systems, and (iv) identifying and proving the system properties that express the specific fault tolerance and mobility-related characteristics of these systems.

The overall project work on the Ambient Campus case study is focused on:

- elucidation of the specific fault tolerance and modelling techniques appropriate for AmI application domain,
- validation of the methodology developed in WP2 and the model checking plug-in for verification based on partial-order reductions, and
- documentation of the experience in the forms of guidelines and fault tolerance templates.

More specifically, in this case study we are investigating how to use formal methods combined with advanced fault tolerance techniques in developing highly dependable AmI applications. In particular, we are developing modelling and design templates for fault tolerant, adaptable and reconfigurable software. The case study covers the development of several working ambient applications (referred to as scenarios) supporting various educational and research activities.

During the second year, our work has been focused on the following major subtask (see Project Description of Work [6.25]):

> **T1.5.4.** Investigate the use of a refinement-based approach to develop a chosen part of the system. Investigate problems specific to model checking based verification of ambient applications.

We have been mainly working on the first scenario – the Ambient Lecture scenario. The following strands of work have been carried out:

- development of the decomposition patterns to be used for stepwise rigorous design of complex fault tolerant AmI systems using the B method (T2.1, T2.3, T2.4)

- stepwise rigorous development of the fault tolerance distributed AmI middleware (T2.1, T2.3)
- development of a method for modelling and model-checking AmI systems and application of the standalone tools for model checking fault tolerance properties (T2.4, T4.2)
- design and implementation of the lecture scenario demonstrator including the CAMA middleware (see sections 6.2.1.1 and 6.2.5) and a lecture scenario application.

6.2. Major Directions of Our Work

6.2.1. Methodology

The Ambient Campus case study directly contributes to RODIN Work Package 2. As a matter of fact, these two strands of work are inseparable: a number of the methods are initially conceived in our work on the case study as this work allows us to grasp the main specific characteristics of the ambient systems and to understand better what methodological advances are needed.

During year 2, we focused our work on defining a set of abstractions to be used in developing and modelling the Ambient Lecture scenarios. We developed a framework called CAMA (see section 6.2.1.1), which consists of a set of fundamental abstractions being used in formal development of ambient systems, in verification of properties of their models and in implementation of these systems.

We formally developed a distributed middleware supporting these abstractions – this was an important development as we firstly demonstrated the applicability of the RODIN methods to developing such types of distributed environments, secondly showed how such systems can be developed starting with a well defined set of abstractions and thirdly ensured in some degree the correctness of our middleware.

Another aim of our work during year 2 was to define a set of formal decomposition patterns for assisting in the stepwise development of the Ambient Campus scenarios. We are applying these patterns now in developing the first scenario – the Ambient Lecture.

6.2.1.1. CAMA Abstractions

CAMA (*C*ontext-*A*ware *M*obile *A*gents) is a middleware supporting rapid development of mobile agent software. It offers a number of high-level operations and a set of abstractions which help programmers to develop multi-agent applications in a disciplined and structured way [6.2].

Agents cooperation in CAMA is based on the concept of *coordination space*. A special entity called *location* provides coordination services to agents. One of the major contributions of CAMA is a novel mechanism to structure coordination space so that groups of communicating agents can work in isolated sub-spaces, called *scopes*. Isolation of a communication space is only one of several roles of the scope construct; for example, it also provides a dynamic type-checking facility for multi-agent applications.

The main structuring units of CAMA applications are *agents* which are pieces of software conforming to some formal *specification*. To distinguish between various functionalities of individual agents, and to match compatible agents, CAMA uses agent *roles* as units of functionality structuring.

Agents are executed on *platforms*, and several agents may reside on a single platform. Each platform provides an execution environment for the agents residing on it, and an interface to the location middleware.

The CAMA middleware architecture was formally modelled using the B Method [6.10]. This allowed us to verify the properties of the scoping mechanism and the ability of agents to tolerate disconnections. The result of the modelling activity was used to implement various parts of the middleware, most notably the scope-related operations.

Exception Handling
Exception handling has proved to be the most general fault tolerance technique as it allows effective application-specific recovery [6.4]. If exception handling is to make programmer's work more productive and less error-prone, the programming and execution environments need to provide adequate support to it. To support exception handling, CAMA introduces inter-agent exception propagation. The mechanism of propagation is complementary to the application-level exception handling. Recovery actions are implemented by application-specific handlers attached to code regions using an exception handling support of the programming language. The task of the propagation mechanism is to transfer exceptions between agents in a reliable and secure way. The enormous freedom of agent behaviour does not allow any guarantees of reliable exception delivery in a general case. We attempt to identify situations where exceptions may be lost or not delivered within a predictable time period.

There are three basic operations that agents can use to send and receive inter-agent exceptions. The first operation, `raise`, propagates an exception to an agent or a scope. The important requirement is that the sending agent – prior to raising an exception – must receive a message from the destination agent (this enables a directed yet anonymous communication between the two agents) and they both must be in the same scope. This operation has two variants:

- `raise(m, e)` - raises exception `e` as a reaction to message `m`. The message is used to trace the producer and to deliver the exception to it. The operation fails if the destination agent has already left the scope in which the message was produced.
- `raise(s, e)` - raises exception `e` to all the participants of the scope `s`.

The crucial requirement to the propagation mechanism is to preserve all the essential properties of agent systems such as agent anonymity, dynamicity and openness. The exception propagation mechanism does not violate the agent anonymity since we prevent the disclosure of agent name at any stage of the propagation process. Note that the `raise` operation does not deal with agent name. Moreover, we guarantee that our propagation method cannot be used to learn the names of other agents.

The other two operations, `check` and `wait`, are used to explicitly poll and wait for inter-agent exceptions:

- `check` - polls for pending exceptions and throws the oldest one (if there are several pending exceptions, only one – the oldest – will be thrown). The `check` operation will not do anything if there is no exception pending for the agent.
- `wait` - waits until any inter-agent exception appears for the agent and raises it in the same way as the `check` operation.

More details on the exception handling provided by CAMA can be found in [6.11].

6.2.1.2. Development patterns and decomposition

The CAMA middleware supports execution of large-scale open agent systems. To facilitate the construction of such systems, we have developed a methodology based on the application of formal methods. The proposed approach addresses a number of challenges, such as interoperability, decentralised development and code reusability.

Agent development commences with a formal specification of the agent's role. All information about the requirements of a role's functionality is contained within a role specification. As long as an agent developer adheres to this specification, it is guaranteed – by the development method – that the role is compatible with all other roles of an application. Many aspects of interaction, such as message ordering and message semantics, which are usually left out in a component interface description are included into a formal role specification. There is no need to analyse the specifications of other roles or to have some additional description of a role in order to build an agent. A role specification is therefore the only information that an agent developer needs for creating an agent.

Our methodology is based on four design patterns:
- The *specification pattern* presents a number of requirements to a specification representing a scope activity;
- The *decoupling pattern* eliminates global scope variables;
- The *refinement pattern* introduces a communication mechanisms; and
- The *decomposition pattern* decomposes a specification of a scope activity into separate role specifications that can be used for implementing collaborating agents.

The specification pattern prescribes the formal design of a CAMA scope specification. A scope specification defines the scope state (program variables) and the scope events (operations) that can affect the scope. The pattern dictates the partitioning of the scope state by distributing program variables among the roles involved so that for each variable, there is exactly one role responsible for updating it. Similarly, operations of a scope are specified in such a way that each operation updates variables of only one role. A scope operation is permitted to read the variables of another role so that two roles can coordinate their activities. As a result, scope operations can be partitioned into the scope roles.

The purpose of the decoupling pattern is to remove any references to external role variables in role reactions. At this stage, we have to ensure that no operations or variables belong to two different roles. The result of the decoupling process is a set of operations that update only the variables of the associated roles. The operations can have a read access to the variables of other roles. The refinement pattern eliminates this last bond between roles.

The refinement pattern replaces the read access to external variables (variables of a different role) with a communication mechanism. The communication mechanism is modelled by the sending and receiving of packets containing a request identification and a number of arguments. The model gives a fair representation of asynchronous communication through the use of message queues. Each role has its own buffer for incoming requests. A role sends a request to another role by storing the request in the incoming buffer of the other role. For simplicity, it is assumed that an implicitly present middleware is responsible for delivering these requests. The model can be easily extended (and thus made more realistic) by introducing separate buffers for incoming and outgoing requests. In the extended model, the middleware would be explicitly modelled as an additional entity that constantly monitors situation and transfers requests from the output buffer of one role into the input buffer of another role.

As a result of introducing communication between role instances, the operations belonging to different roles become completely decoupled. The only way that one role instance can affect another is via the communication mechanism. This permits us to decompose the scope specification as well as to separate the program variables and operations of different roles and put them into different role specifications. The refinement pattern used at this step is based on the idea of the B decomposition refinement [6.19].

6.2.2. Plugins

An integral part of our work on the Ambient Campus case study is an investigation into the problem of verifying the correctness of mobility designs, including fault-tolerant mechanisms, expressed in notations such as CAMA. It is widely recognised that this is a complex problem, and one way of addressing it is by using model-checking [6.3] which is completely automatic, relatively fast compared to other alternatives, and produces counterexamples which can be used for debugging. It is therefore particularly attractive in an industrial context [6.24].

Model-checking carries out the verification of a system using a finite representation of its state space, and exhibits a trade-off between the compactness of this representation and time efficiency. For example, deadlock detection is PSPACE-complete for a compact (bounded) Petri net or equivalent process algebra representation, but polynomial for state graph representation. However, the latter is often exponentially larger, causing the *state space explosion* problem [6.26].

Mobile systems are highly concurrent causing a state space explosion when applying model-checking techniques. One should therefore use an approach which alleviates this problem. In our case, we focus on an approach based on partial order semantics of concurrency and the corresponding Petri net unfoldings [6.18]. A finite and complete unfolding prefix of a Petri net is a finite acyclic net which implicitly represents all the reachable states of the original net. Efficient algorithms exist for building such prefixes [6.12], and complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent systems, because they represent concurrency directly rather than by multidimensional "diamonds" as is done in state graphs. Referring to the mobility model of CAMA, our approach is particularly suitable for verification of *reachability-like* (or *state*) properties, such as:

- The system never deadlocks, though it may terminate in a pre-defined set of successful termination states.
- Security properties, i.e., all sub-scope participants are participants of the containing scope.
- Proper using of the scoping mechanism, for example: a scope owner does not attempt to leave without removing the scope; agents do not leave or delete a scope when other agents expect some input from the scope; and the owner of a scope does not delete it while there are still active agents in the scope.
- Proper use of cooperative recovery: all scope exceptions must be handled when a scope completes; all scope participants eventually complete exception handling; and no exceptions are raised in a scope after an agent leaves it.
- Application-specific invariants. (Note that the negation of an invariant is a state property, i.e., the invariant holds iff there is no reachable state of the system where it is violated.)

To our knowledge, this is the first attempt to develop unfolding based model checking technique for mobile systems, and consequently it was clear right from the start that we would need to make several decisions of both theoretical and implementation nature.

6.2.2.1. Our approach

The first problem we had to address was the choice of formal model for capturing the properties and behaviour of mobile systems. Since the development of mobility plugins proceeded concurrently with other work on the Ambient Campus case study, a decision was made to focus on existing formalisms for mobility with the view that the expected model checkers would then be easily adaptable to the context of this case study. We decided to investigate two process algebras for mobility, viz. π-calculus [6.20, 6.23] and KLAIM [6.21, 6.22]. They represent both synchronous and asynchronous models of distributed systems computation, and so we expected that they would cover a full range of issues relating to the Ambient Campus case study. π-calculus allows, in particular, to express dynamic change in the process ability to communicate with the external environment, by passing new *channels* through interactions on previously known channels. It also provides means to express and reason about a variety of security related aspects. The choice of the second model was influenced by our work in other parts of the Ambient Campus case study. KLAIM, in particular, supports explicit localities which are first-class data that can be manipulated by active processes, and coordination primitives for controlled interactions among processes located at network's localities.

We set out to develop semantics-preserving translations (in the sense of provably generating strongly equivalent labelled transition systems) of both the π-calculus and KLAIM into suitable classes of high-level Petri nets. Our choice for the latter was a modular model of high-level Petri nets, resulting in compositional translation of process expression terms. The work on translating π-calculus has been carried out throughout the duration of RODIN, and resulted in two translations: one for finite π-calculus terms [6.8], and recently for general recursive expressions [6.6]. The translation of KLAIM was based on some key ideas of our work on the π-calculus, and was first reported in [6.5] and then in [6.7].

There are several variations of the unfolding-based model-checking technique, and the Petri nets produced by our translations suggests that one should employ a variant which is capable to

deal efficiently with coloured tokens and high-level arc annotations. In particular, one should avoid the expansion of the high-level Petri net to its low-level representation and unfolding the latter, since such an approach may yield a huge intermediate low-level net, rendering the whole attempt practically useless (see [6.14] for a discussion of this issue). We therefore decided to use the PUNF model-checker (complemented by the MPSAT property verifier based), which directly applies unfolding to high-level nets without expanding them to low-level nets [6.14].

We developed two prototype tools which required a significant effort. The first problem was that Petri nets resulting from the translations do not conform to the input format required by the unfolder (two particular issues here are the read arcs and non-safe places of the high-level nets). Another problem concerned the infinite branching in the operational semantics of process expressions. The tool presented in [6.16, 6.17] is suitable for the task of model-checking of finite π-calculus terms, implementing the translation of [6.8]. To lift it to a suitable subset the recursive (or iterative) case described in [6.6] is a subject of ongoing investigations. The tool presented in [6.15] is suitable for the task of model-checking of KLAIM terms generating finite state space, implementing the translation of [6.5].

6.2.2.2. Examples and experimental results

In our experiments described in [6.16, 6.5], we used simple 'classroom' scenarios inspired by the Ambient Campus case study. A typical π-calculus-like example has the following form:

$$\text{NESS}(n) \stackrel{\text{df}}{=} (\boldsymbol{\nu} h)(\boldsymbol{\nu} h_1) \ldots (\boldsymbol{\nu} h_n)\ (T|S_1| \ldots |S_n)$$

where T represents a 'teacher' process, and each S_i a 'student' process. Their respective definitions are as follows (note that input prefixes are denoted as $a?b$ and the output ones as $a!b$):

$$T \stackrel{\text{df}}{=} a?ness\,.\,(h_1!ness\,.\,h_1?x_1\,.\,0 \mid \cdots \mid h_n!ness\,.\,h_n?x_n\,.\,0)$$

$$S_i \stackrel{\text{df}}{=} h_i?addr_i\,.\,(h!h_i\,.\,h_i!done\,.\,0 + h?another_i\,.\,addr_i!h_i\,.\,addr_i!another_i\,.\,h_i!done\,.\,0)$$

The idea is that the teacher first receives from the school electronic submission system[1] a channel $ness$ using which the students are supposed to submit their work for assessment. The teacher passes this channel to all the students (using n parallel sub-processes), and (also in parallel) then waits for the confirmation that the students have finished working on the assignment before terminating. A student's behaviour is somewhat more complicated. After receiving the $ness$ channel, students are supposed to organise themselves to work on the assignment in pairs and, after finishing, exactly one of them sends to the support system (using the previously acquired $ness$ channel) two channels which give access to their completed joint work. The students finally notify the teacher about the completion of their work. The property to verify is that all the processes involved in the computation successfully terminate by reaching the end of their individual code. For instance, the following move is possible for the initial expression:

$$(\boldsymbol{\nu} h)(\boldsymbol{\nu} h_1) \ldots (\boldsymbol{\nu} h_n)\ (T|S_1| \ldots |S_n) \xrightarrow{\overline{a}b} (\boldsymbol{\nu} h)(\boldsymbol{\nu} h_1) \ldots (\boldsymbol{\nu} h_n)\ (T'|S_1| \ldots |S_n)$$

where b is the channel on which links to the completed pieces of coursework are to be submitted, and $T' \stackrel{\text{df}}{=} h_1!b\,.\,h_1?x_1\,.\,0 \mid \cdots \mid h_n!b\,.\,h_n?x_n\,.\,0$.

[1] Called NESS in Newcastle.

Examples like that described above allowed us to have easily scalable specifications, which satisfy a correctness property only for some values of n. (Note that for the above example only even n leads to a successful termination). Also, the examples were interesting by exhibiting *different* sources of state space explosion, e.g., coming from parallel composition and choice constructs. The former kind of state space explosion is typically avoided by the unfolding based model-checking techniques, whereas techniques based on interleaving suffer from it. To treat the latter kind of state space explosion we have initiated work on a highly promising novel unfolding-based technique [6.13].

The work on Ambient Campus scenarios resulted in the discovery of further possible improvements to the way in which process algebra expressions are translated into Petri nets. Crucially, the experiments conducted so far — though still limited in their number and scope — are highly encouraging and seem to confirm our initial hypothesis that unfolding based model checking is a promising verification technique for the verification mobile computing systems.

6.2.3. Year 2 demonstration

To present the project work on CS5 we are working on a demonstration which overviews the main strands of work in this area and follows the script below.

The demonstration aims to show a system implemented using the CAMA approach to support lecture activities. In the following part of the demonstration, we will discuss some of the RODIN methods and tools which are being developed by the project team to support rigorous design of the ambient systems (see sections 6.2.1 and 6.2.2).

The system runs on the CAMA middleware distributed over several students' devices – such as PDAs, smartphones, or laptops – and a lecturer's desktop computer. The lecturer will be able to register students when the lecture starts, create groups and assign students to these groups, monitor the group work and assist groups when necessary. We focus on a specific lecture activity where students are organised into groups and they learn together the B method through a supporting B tool. The full description of this activity can be found in section 6.2.4.

During this demonstration, the main functionality of the system will be shown, including distributed editing of the assigned B model by group members, distributed chat/discussion among the group members, sending of developed models to the prover, interacting with the prover to input manual proofs, as well as teacher's monitoring-of and assisting-to the group work.

At the next step of the demonstration, we will show the formal development of the Ambient Lecture scenario using *decomposition refinement patterns*, ensuring fault tolerance and interoperability of the system components (called agents in our work) developed by different independent programmers. This formal stepwise development process supports the construction of systems using a set of abstractions, such as roles, agents, locations, scopes and platforms (see section 6.2.1.1). We will demonstrate formal stepwise development of the distributed CAMA middleware on which the lecture scenario runs. This development has guided our implementation of the middleware. This work has been conducted to ensure the correctness of the platform and to demonstrate the applicability of the RODIN method in this new challenging area.

The last step of the demonstration will show the application of a standalone prototype of the RODIN mobility plugin for developing the lecture scenario. In particular, we will show how several fault tolerance related properties of the application have been model-checked. This work has allowed us to give feedback to the development of the plugin, to gain some initial experience in combining stepwise refinement with model checking, and to better understand the modelling language suitable for both state-based and process-based reasoning about the ambient systems.

6.2.4. Description of the lecture scenario

To distinguish between the "traditional lecture setting" (defined in D2, D4 and D8) and the the "software-assisted lecture setting" (outlined in this section), we refer to the latter as *Ambient Lecture*.

The Ambient Lecture system is being designed to meet the requirements set out in [6.1]. In this design, each classroom is a location with a wireless support, in which a lecture is conducted. An agent can take one of the two roles: teacher or student. The teacher agent runs on a desktop computer available in the classroom, while student agents are executed on PDAs (each student is given a PDA).

We use the scoping mechanism described in section 6.2.1 to structure the system. The teacher agent creates the outer scope representing the global scope into which student agents join. An Ambient Lecture starts when there is one teacher agent and a predefined number of student agents joining this scope.

To support better system structuring, data and behaviour encapsulation, as well as fault tolerance, all major activities during the Ambient Lecture are conducted within subscopes (nested scopes). The group work is one of the activities performed within a nested scope. Teacher – through his/her agent – arranges students into groups, so that only students belonging to the same group can communicate with each other through their agent. Each group is then given a task to solve – in this case, a B specification. Students within the same group work together on the solution and present their answer at the end of the group work stage. Teacher agent monitors the communication in all groups so that if necessary, the teacher can give guidance to assist the students to complete their task.

At the beginning of any Ambient Lecture, all agents (teacher and students alike) are placed in the global scope. The teacher agent keeps a list of all students joining the Ambient Lecture, and through the application's graphical user interface (GUI), the teacher can select which students to be placed within each group. Each group is given a unique name and the groups are mutually exclusive, i.e. a student cannot belong to more than one group. The teacher agent creates a subscope for each group and issues a *StartGroup* instruction to the student agents involved so that they automatically join the subscope they are assigned to. This is achieved by executing the CAMA `JoinScope` operation that uses the group name as a parameter. This structuring guarantees that while within a group, a student can only send messages to other students belonging to the same group, but he/she will also receive any message sent in the global lecture scope.

The following subsesctions illustrate the operations that can be carried out by both teacher and student agents during the Ambient Lecture.

6.2.4.1. Initialisation of an Ambient Lecture

This is automatically performed (by the teacher agent) when the teacher invokes the teacher agent software. Actions performed include the creation of the global Ambient Lecture scope based on the scope restrictions (which dictate the type and the number of agents allowed to join).

6.2.4.2. Joining and registration into an Ambient Lecture

An agent must join an Ambient Lecture before they can participate fully in it. Teacher agent will automatically join the Ambient Lecture straight after creating it. Student agent will need to explicitly join the Ambient Lecture available in the location. This will also trigger the registration process that can be observed through the teacher agent.

Teacher

- *Registers each student and starts the lecture*
  Teacher agent keeps a record of student agents currently joining the Ambient Lecture. This allows the teacher, for example, to assign the students into groups later during the lecture.

Student

- *Joins an Ambient Lecture and waits until the lecture starts*
  Two modes of operation could be employed here. One is where no interaction is allowed until the scope restrictions are fulfilled (e.g. concerning the number of student agents that must already be present). Another is when student agents can start interacting straight away with whoever is already in the scope.

6.2.4.3. During Ambient Lecture

Students can join the Ambient Lecture anytime during the lecture, as long as there is still a space available to them. As soon as there are enough student agents in the Ambient Lecture scope, the Ambient Lecture can start; more students may join later. Students can only join with the same agent once at the same time.

Teacher

- *Sends messages to students*
  Teacher may broadcast messages to all students. At a later implementation, we may add a feature to allow teacher to send a direct (private) message directly to a particular student during the Ambient Lecture.
- *Organises students into groups*
  The Ambient Lecture software allows teacher to put students into groups. The agents of students belonging to the same group will reside in the same subscope, hence enabling group communication and collaborative group work.

- *Sends message to groups*
  Teacher may send messages to students in a particular group during group work. This will be a broadcast to the group subscope.
- *Receives questions from students*
  Students may ask questions during the lecture. These questions may be raised verbally or through the Ambient Lecture chat software.

Student

- *Chats with other students*
  Communication among students (and teacher) is conducted through a chat window where messages are typed, broadcast and received in real time. When in a group scope (see below), students can choose whether to send the messages just within the group or to the global Ambient Lecture scope.
- *Asks teacher questions and sees replies from him*
  There are two modes for this interaction. The first one is where student broadcasts the question to everyone in their current scope (could be in the group scope) and teacher's reply comes as a broadcast as well. The second mode allows student to send the question privately to the teacher and the teacher can choose whether to respond privately as well or to broadcast the answer to a group or the whole class. At the moment, only the first mode is implemented.
- *Joins a group*
  Student agents – upon request from the teacher agent – join a subscope of the Ambient Lecture scope in order to carry out group work. Each student agent can only be in one subscope at any time.
- *Leaves Ambient Lecture*
  Students can leave the Ambient Lecture anytime. They should be able to rejoin later, as long as the Ambient Lecture setting is still running and there is a space available. If a student leaves the Ambient Lecture while in a group work, its agent will also leave the group subscope. This agent will not go back to the same group subscope if it rejoins the Ambient Lecture.

6.2.4.4. During group work

By default, student agents joining an Ambient Lecture will be in the global lecture scope. From time to time, teacher will organise students into group, which means that the corresponding student agents will join a subscope allocated for this group. Each student agent can only be in one subscope (on top of being in the global lecture scope) at any one time.

Teacher

Teacher prepares the group work by organising students into groups, assigning a B-project for each group to work on, and monitoring each group.

- *Assigns a B-project to a group*
  Each group will be given a B-project to work on, which contains at least one B-machine specification that the students need to edit and run B-commands on.

- *Watches activity of each student*
  This monitoring activity is useful to measure each student's participation during the group work. A passive student might require further help or different group arrangements might be needed.
- *Inspects edited files*
  Teacher can check the progress of the group work by inspecting the changes that the students made on the files and by checking the status of the B-commands already issued.
- *Assists by editing files*
  Teacher may modify the B-machine specification files in order to make it clearer for the students how to solve the problem, or to "reset" the file if the students made too many mistakes.
- *Takes part in discussion*
  Teacher may help students to understand the problem they are trying to solve by asking probing questions as well as giving hints and advice.
- *Forces unlocking of resources*
  If a student appears to hold a file for too long (this could happen, say if the student agent crashes), the teacher can manually unlock the file to allow other students to edit it.

Student

Students' actions during group work mostly concern with editing B-machine specification and carrying out the B-commands such as proving and type checking. They can also communicate with other student agents within their group, the teacher as well other student agents in the global lecture scope. We are thinking about disabling the communication with other student agents in the global scope.

- *Chooses file to work on within a project*
  Each project will have a list of associated files, and the student can choose which file to work on. This file represents a B-machine specification and each student is allowed to work with only one file at a time.
- *Edits a file*
  There is a *shared editor* window that provides concurrency control (multiple readers, one writer) for editing a file. A student agent needs to obtain a lock before it can edit a file. Only one agent can edit each file at any one time, although other agents can read the content of this file and see the update in real time. The lock must be released by the writing agent upon the completion of the editing process.
- *Proves/model checks/type checks/does interactive proving*
  The Ambient Lecture software allows students to carry out these commands on the B-machine specification they are working with. With the current implementation, the student agents are not required to obtain the editing lock first before carrying out these commands. We agree that this is not a desirable feature, and we will fix this in the later implementation.
- *Takes part in discussion*
  During the discussion, students may ask questions, and other students in the

group may provide the answer. If the questions remain unanswered, the group may ask teacher for assistance.

- *Asks teacher assistance*
  Teacher monitors group work, and from time to time, students may ask teacher for clarification on the task they are working at.
- *Sends message to other students*
  Students can send messages to other students in the same group.

Students cannot explicitly leave a group; only teacher can decide whether a student must leave a group, for example at the end of the group work. Students can leave the Ambient Lecture setting altogether though, and when this happens, they will automatically leave the group subscope as well.

We have implemented and tested the Ambient Lecture application supporting all of the operations mentioned above. At this stage, our testing only involves a small number of agents running on a desktop computer, laptops and PDAs. We plan to carry out a more thorough testing with greater number of agents and we will investigate the feasibility of adding more operations. There are still areas where our application can be improved; these are outlined in section 6.3.

6.2.5. CAMA middleware architecture and ongoing implementation

In the current version of the CAMA system [6.2], the location middleware is implemented in C (we call it $_c$CAMA). This allows us to achieve the best possible performance of the coordination space and to effectively implement numerous extension, such as the scoping mechanism. The location middleware implementation is quite compact - it consists of approximately 6000 lines of C code and should run on most Unix platforms. We have so far tested it on Linux FC2 and Solaris 10.

In order to use the location middleware, we have also developed a CAMA adaptation layer in Java[2] called $_j$CAMA. This adaptation layer defines several classes for representing – among others – the abstract notions of Location, Scope and Linda coordination primitives. $_j$CAMA provides an interface through which mobile agents or applications can be developed easily.

The full implementation of the location middleware and the adaptation layer are available at SourceForge [6.9].

A diagrammatic representation of the CAMA-based system architecture can be seen in Figure 6.1. Each platform carries a copy of $_j$CAMA. Agents residing on a platform uses the features provided by $_j$CAMA to connect over the wireless network to the $_c$CAMA location middleware.

It is possible to construct adaptation layers for other platforms and languages. For now, the $_j$CAMA Java adaptation layer outlined above permits agent development for PocketPC-based PDAs. It has a very small footprint (˜60Kb) and can be used with both standard Java and

[2]We use Java for developing the applications for PDAs.

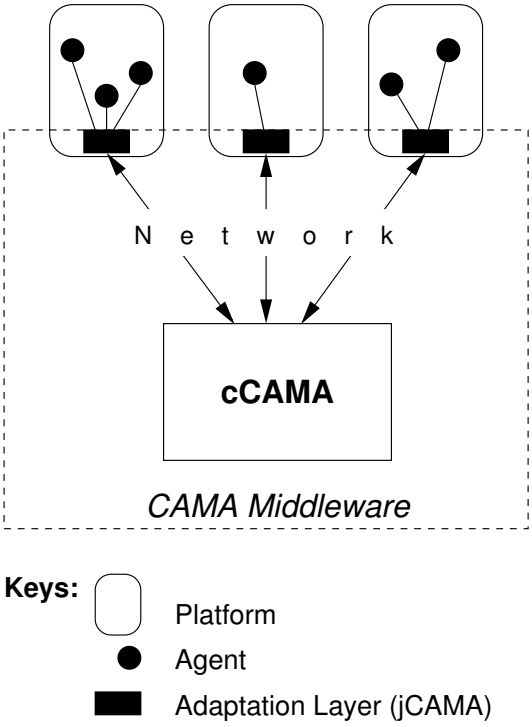


Figure 6.1. CAMA architecture

J2ME. In the future, we plan to develop adaptation layers for other languages such as Python and Visual Basic, as well as versions compatible for smartphone devices.

6.3. Summary and Future Work

6.3.1. Ambient Campus Group Project

This project is to be undertaken by postgraduate students on an advanced MSc course (SDIA – System Design for Internet Applications) at the School of Computing Science of Newcastle University. From the point of view of the Ambient Campus case study, its main aims are (i) to provide a testing environment for the fault tolerant mechanisms and software developed within the Ambient Campus case study; and (ii) to experiment with the methodologies for developing fault-tolerant mobility code investigated within the Ambient Campus case study.

The class of approximately 24 students will be split into four groups: A, B, C and D. Groups A and B will be issued with PDAs, while groups C and D will work without the PDAs. Groups A and C will use the methodologies for developing fault-tolerant mobility code investigated within the Ambient Campus case study, while groups B and D will not use these.

Each group is asked to develop a software application involving mobility and context awareness, and it needs to handle potential faults. One example of such an application is a system for guiding a newly arrived student through the registration process during an induction week. This application would allow one to include both individual physical mobility aspects (such as

finding a way to the University Registration Desk or the office of Graduate School), as well as interaction with staff and other students.

The idea is to provide support for the different phases of the group project, and the different roles the participating students will play over that period; starting from conducting research and finishing it with the preparations for the final presentations. A number of standard small applications will be provided, such as those for organising/conducting a meeting and for electing a group leader. PDAs (and software running on them) will provide a basic support for organising meetings, exchanging ideas and quick comments, informing about ongoing research, etc. The software running on the desktop computers will provide support for carrying out programming task, such as in exchanging and integrating code, as well as for system testing.

6.3.2. Switching to Event-B and the new platform

Until recently, we have been using AtelierB toolkit and the classical B in the development of this case study. However we tried to limit ourselves to the specification style supported by Event-B [6.19]. For the middleware development and the work on methodology, we used the *B to Event-B* translator. This simplifies the transition to Event-B and the new RODIN platform. The formal agent development methodology described above is therefore based on Event-B. A number of features specific to Event-B, such as atomicity refinement, variants and decomposition, are extensively used in the middleware and during an agent design.

The approach used in the modelling of the Ambient Campus scenario is based on the reactions architecture, which fits perfectly to the style of Event-B specifications. Atomically executed Event-B events are close to the asynchronously executed reactions. This is crucial for building agent implementations from formal role specifications.

We plan to conduct some initial experiments with the new RODIN platform in late September-October when the new version of the platform supporting refinement is released. In particular, we plan to formally model the case study using the new platform applying the formal design methodology.

6.3.3. Future work on Case Study demonstrators

Our main work in the coming year will focus on finalising the CAMA modelling notations, conducting Ambient Campus Group Project experiments, developing the second Ambient Campus scenario (most likely, supporting distributed student group work), gaining extensive experience in using the RODIN platform and the mobility plug-in, capturing this experience in developing Ambient Campus scenarios and extracting it in the forms of reusable development patterns and on developing the final RODIN demonstration.

6.4. References

6.1 B. Arief, J. Coleman, A. Hall, A. Hilton, A. Iliasov, I. Johnson, C. Jones, L. Laibinis, S. Leppanen, I. Oliver, A. Romanovsky, C. Snook, E. Troubitsyna, and J. Ziegler. Rodin Deliverable D4: Traceable Requirements Document for Case Studies. Techni-

cal report, Project IST-511599, School of Computing Science, University of Newcastle, 2005.

6.2 B. Arief, A. Iliasov, and A. Romanovsky. On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems. In *Proceedings of the 5th International Workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS) at ICSE 2006*, pages 29–36, 22-23 May 2006.

6.3 E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

6.4 F. Cristian. Exception Handling and Fault Tolerance of Software Faults. In M. Lyu, editor, *Software Fault Tolerance*, pages 81–107. Wiley, NY, 1995.

6.5 R. Devillers, H. Klaudel, and M. Koutny. A Petri Net Semantics of a Simple Process Algebra for Mobility. Technical report, CS-TR: 912, School of Computing Science, University of Newcastle, 2005.

6.6 R. Devillers, H. Klaudel, and M. Koutny. A Petri net translation of pi-calculus terms. Technical report, CS-TR: 887, School of Computing Science, University of Newcastle, 2005.

6.7 R. Devillers, H. Klaudel, and M. Koutny. Formal modeling and quantitive analysis of KLAIM-based mobile systems. In *EXPRESS 2005*, 2005.

6.8 R. Devillers, H. Klaudel, and M. Koutny. Petri Net Semantics of the Finite pi-calculus Terms. *Fundamenta Informaticae*, 70:203–226, 2006.

6.9 A. Iliasov. Implementation of Cama Middleware. http://sourceforge.net/projects/cama [Last accessed: 1 Feb 2006].

6.10 A. Iliasov, L. Laibinis, A. Romanovsky, and E. Troubitsyna. Rigorous Development of Fault Tolerant Agent Systems. Technical report, Number 762, Turku Centre for Computer Science, 2006.

6.11 A. Iliasov and A. Romanovsky. CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents. Presented at ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions, 25 July 2005, Glasgow, UK, 2005.

6.12 V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Computing Science, University of Newcastle upon Tyne, 2003.

6.13 V. Khomenko, A. Kondratyev, M. Koutny, and V. Vogler. Merged Processes — a New Condensed Representation of Petri Net Behaviour. In *CONCUR 2005*, volume 3653 of *Lecture Notes in Computer Science*, pages 338–352, 2005.

6.14 V. Khomenko and M. Koutny. Branching Processes of High-Level Petri Nets. In *TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*, pages 458–472, 2003.

6.15 V. Khomenko, A. Niaouris, and M. Koutny. Applying Petri Net Unfoldings for Verification of Klaim Expressions. Technical report, School of Computing Science, University of Newcastle, 2006.

6.16 V. Khomenko, A. Niaouris, and M. Koutny. Applying Petri Net Unfoldings for Verification of Mobile Systems. Technical report, CS-TR: 953, School of Computing Science, University of Newcastle, 2006.

6.17 V. Khomenko, A. Niaouris, and M. Koutny. Applying Petri Net Unfoldings for Verification of Mobile Systems. In *MOCA 2006*, 2006.

6.18 K. McMillan. Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. In *CAV 1992*, volume 663 of *Lecture Notes in Computer*

Science, pages 164–174, 1992.

6.19 C. Metayer, J.-R. Abrial, and L. Voisin. Rodin Deliverable 3.2: Event-B Language. Technical report, Project IST-511599, School of Computing Science, University of Newcastle, 31 May 2005.

6.20 R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Control*, 100:1–77, 1992.

6.21 R. D. Nicola, G. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

6.22 R. D. Nicola, D. Latella, and M. Massink. Formal modeling and quantitive analysis of KLAIM-based mobile systems. In *Applied Computing*, pages 428–435, 2005.

6.23 J. Parrow. An Introduction to the π-calculus. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

6.24 C. Pixley. Formal Verification in 2004. In *DATE 2004, EDA Tools Forum*, 2004.

6.25 Rigorous Open Development Environment for Complex Systems (RODIN), Description of Work. *IST 6^{th} Framework Programme, Proposal No. 511599*, April 2004.

6.26 A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.