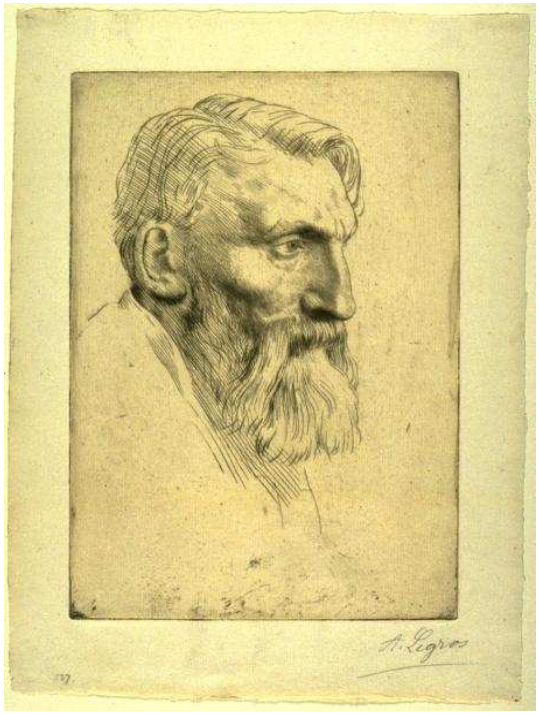
Information Society Technologies

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



RODIN Deliverable D8

Initial Report on Case Study Development

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

Public Document

31st August 2005

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

Contributors:

Peter Amey (Praxis High Integrity Systems Ltd, UK),
Joey Coleman (University of Newcastle upon Tyne, UK),
Budi Arief (University of Newcastle upon Tyne, UK),
Neil Evans (University of Southampton, UK),
Alex Iliasov (University of Newcastle upon Tyne, UK),
Ian Johnson (ATEC Engine Controls Ltd, UK),
Maciej Koutny(University of Newcastle upon Tyne, UK)
Linas Laibinis (Aabo Akademi University, Finland),
Sari Leppänen (Nokia, Finland),
Ian Oliver (Nokia, Finland),
Alexander Romanovsky (University of Newcastle upon Tyne, UK),
Colin Snook (University of Southampton, UK),
Elena Troubitsyna (Aabo Akademi University, Finland),

CONTENTS

SECTION 1. INTRODUCTION

This document reports on the first year of the development of case studies in RODIN. The aim of case studies is to drive the development of RODIN methodology and supporting platform, validate it and evaluate its cost-effectiveness. While the deliverable D14 gives the precise assessment of the case study developments, in this document we focus mostly on describing the achieved results and outline the future plans.

In general, we believe that the work on the case studies is proceeding as planned. Within each case study we have identified challenging research topics to be addressed in RODIN methodology. While working on case studies we explored the corresponding domain specific models and problems. This allowed us to identify the additional requirements on supporting tool platform. Hence diversity of case studies has facilitated achieving versatility of the supporting platform.

The description of the achieved results and the outline of the future development for the case study 1 – *Formal Approaches to Protocol Engineering* – is presented in Section 2. This case study investigates the use of formal methods in the development of communicating systems. Its major goal is to explore an incorporation of formal methods into existing UML-based development cycle. The advances in creating a methodology for formalizing UML-based development and identifying the requirements for tool support are reported in this document.

Section 3 is devoted to the evaluation of the work done within case study 2 – *Engine Failure Management System*. The methods and tools developed in RODIN should potentially improve maintenance and re-use of the failure management systems. This task is tackled from two directions. The first direction investigates an accurate modelling of the domain to reduce the semantic gap between application requirements and systems design. The second direction explores how to promote reusability by developing configurable generic specifications. In this document we report on the current state of research on this topic and future plans.

In Section 4, the initial results of the work on the case study 4 – *Formal Techniques within an MDA Context* are presented. This case study is primarily aimed at constructing and verifying platforms or architectures rather than aiming at the applications themselves. The case study explores the problem related to the MDA development flow together with the role of verification/validation within that flow, as well as problems related to development methods used within Nokia. The section provides the background required to understand the scope of the case study, outlines the research directions and evaluates the obtained results.

Section 5 analyses the initial results of the development of case study 4 – *CDIS Air Traffic Control Display System.* This case study aims at testing RODIN methods and tools on a complex, data-intensive and highly-distributed system. The system was successfully developed a decade ago but the development encountered a number of difficulties. The chapter presents the analysis of the challenges to be tackled, evaluates the initial results on addressing them and outlines further directions.

In Section 6 we present the overview of the development of case study 5 -- *Ambient Campus*. The aim of this case study is to investigate modelling, development and verification of fault tolerant mobile asynchronous systems. The work on the case study during the first year has addressed a number of fundamental theoretical issues, such as process algebraic and state-based approaches to modelling mobile systems, modelling fault tolerance, reasoning about correctness of ambient system etc. Moreover, this work has actively contributed to definition of the supporting platform and plug-ins. A brief evaluation of the achieved results and outline of plans are presented in this document.

The case studies have naturally promoted a co-operation between academic and industrial partners and facilitated overall consolidation of the project. A number of joint research efforts have been initiated already during the first year and will be further expanded and strengthened during the next years.

SECTION 2. REPORT ON CASE STUDY DEVELOPMENTS FOR CASE STUDY 1: FORMAL APPROACHES TO PROTOCOL ENGINEERING

2.1. Introduction

The goal of the CS1 – Formal Approaches to Protocol Engineering – is to investigate application of formal methods for development of telecommunication systems and protocols (Project Description of Work [2.1]). The protocol engineering group in Nokia has developed the ”Lyra” method which supports the service-oriented approach to protocol engineering. 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 we have got deeper understanding of the Lyra development method and its application domain. The concrete case study suggested by Nokia, the Third Generation Partnership Project (3GPP) positioning service, facilitated studying the Lyra development process and served as a testbench for emerging formal methodology and tools.

We have started to formalise the Lyra development process in the B Method [2.3]. The B Method is an approach to industrial development of highly dependable software, supporting formal system development by the stepwise refinement method. We have developed the guidelines for translating the Lyra UML2 models into B specifications. Moreover, we have associated the Lyra development phases with the corresponding B development steps – B refinements. This creates the basis for verifying correctness of main development phases of Lyra.

In addition, we have developed a B specification pattern for a communicating service component, which can be understood as a ”building block” of communicating systems in Lyra. The pattern can be recursively used to specify service components on different layers of abstraction. We have also identified and formalised the communicational and functional aspects of such communicating components.

Finally, we have started to tackle fault tolerance and parallelism issues that are inherent for telecommunication systems. We are going to further investigate these issues during the second year of the RODIN project.

2.2. Major Directions in Case Study Development

Methodological issues brought up by the case study. The case study has raised several interesting methodological issues. The main goal of the case study is to achieve automatic translation of the UML2-based development of communicating systems into the

corresponding specification and refinement process in the B Method. Tho achieve this, we need to develop a general methodology supporting automatic translation of UML2-based models into the B specification language. Once the methodology is developed, we will cooperate with the U2B [2.7] tool developers to provide automatic tool support for such translation.

Telecommunication systems operate in volative, error prone environment. The most typical faults are lost or corrupted communication messages between distant network elements. Therefore, fault tolerance should be an intrinsic feature of telecommunication systems. Hence the fault tolerance mechanisms should be integrated into both specification and development process of such systems.

Telecommunication systems provide certain services for the external users. Usually the service execution is distributed over distinct network elements which provide their services as parts of the required service execution. Often these services are executed in parallel. We should take parallelism into account while modelling or verifying telecommunication systems.

The Lyra development is currently validated by using model-based testing of program code or intermediate UML2 models. By translating these UML2 models into the B specifications, we would create additional more precise system models that can be used to facilitate test generation.

Methodological advances used in the case study. We believe that the methodological results achieved during the first year of the project establish a solid basis for creating a methodology for formal development of communicating systems. We have associated the stages of Lyra development with the corresponding B development steps – B refinements.

We have created a B specification pattern for a communicating service component which is a ”building block” of communicating systems in Lyra. The pattern can be used to specify communicating service components at different layers of abstraction, e.g., the components providing single external services or the components orchestrating service execution by relying on other (lower layer) service components.

We have identified communicational and functional parts of communicating components and formalised them in B. Both communicational and functional parts of a communicating components are specified in such a way that they can be instantiated by different concrete communication protocols or calculational algorithms during the development (refinement) process.

We have also developed refinement patterns which formalise the decomposition and distribution steps in the Lyra-B development. The patterns can be instantiated for a specific functional or network architecture. The use of the refinement patterns would allow us to automate verification of the Lyra development process. This would be

achieved by automatic proof of refinements between the corresponding B models by means of the available tool support.

We are currently developing B models of the fault tolerance mechanisms for communicating systems and integrating them into the specification and refinement patterns of communicating components. The fault tolerance mechanisms will allow us to model simple recovery procedures in the cases when a service component failed or a communication message has been lost. More detailed fault tolerance mechanisms will be integrated at some later development step when the details of used communication protocols become available.

We are also in the process of formulating the precise correspondencies between the Lyra UML2 models and the corresponding B specifications. These correspondencies will be used for automatic translation of the Lyra UML2 models into B.

We have started to investigate parallel execution of communicating service components. We can model parallel execution of different services already at the service decomposition phase. Moreover, we are going to verify possible distribution of parallel behaviour over the given network architecture at the service distribution phase.

Impact of the case study on the platform development. As a result of this case study, we are going to develop a collection of patterns for specifying and developing communicating systems.

Impact of the case study on the plug-in development. The case study sets the requirements for the development of the model-based testing plug-in on the basis of experience accumulated at the Nokia research center. The requirements for this plug-in are described in D11. We plan to integrate our efforts with the U2B developers on automatic translation of the Lyra UML2 models into B. Therefore, the case study may have impact on the development of the U2B plug-in.

2.3. Achieved Results

The first year of the RODIN project our work has focused on three major tasks (see Project Description of Work [2.1]):

- **T1.1.1** Define the case study, evaluation plan, measurements and assessment criteria.
- **T1.1.2** Review the ”Lyra” method and identify the development steps, which should be tackled by RODIN. Assign the sets of methods and techniques to be applied at these steps.
- **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.

The main results achieved during the first year are the following:

1. We have prepared the traceable requirement document [2.2] for this case study.
2. We have developed the guidelines for translating the Lyra UML2 models into B specifications.
3. We have associated the decomposition and distribution steps of the Lyra development process with the corresponding B refinement steps.
4. We have developed the B specification and refinement patterns for communicating systems modelled in Lyra.
5. We have started to investigate formal modelling of fault tolerance and parallelism aspects of telecommunication systems.

We present the more detailed results by using the case study suggested by Nokia – the 3GPP positioning system [2.8, 2.9]. In the first part we demonstrate the Lyra UML2-based development of the system. Then we show how we can formalise and validate this Lyra development by using the B Method.

2.3.1 Lyra and 3GPP positioning system

Lyra [2.6] is a model-driven and component-based design method for the development of communicating systems and communication protocols. It has been developed in the Nokia Research Center by integrating the best practices and design patterns established in the area of communicating systems. The method covers all industrial specification and design phases from prestandartisation to final implementation.

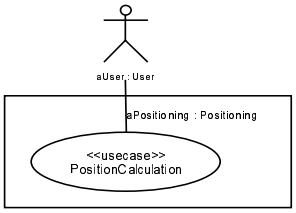
Lyra has four main phases: Service Specification, Service Decomposition, Service Distribution and Service Implementation. The *Service Specification* phase focuses on defining services provided by the system and their users. In the *Service Decomposition* phase the abstract model produced at the previous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

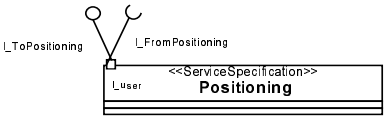
We model part of a Third Generation Partnership Project (3GPP) positioning system [2.8, 2.9]. The positioning system provides positioning services to calculate the physical location of a given user equipment (UE) in a Universal Mobile Telecommunication System (UMTS) network. We focus on Position Calculation Application Part (PCAP) – a part of the positioning system allowing communication in the Radio Access Network (RAN).

As a part of the RODIN project, we have developed the requirements document [2.2] for the 3GPP positioning system. The document presents traceable requirements classified into 3 categories: architectural, functional, and communicational. In addition, the functional requirements for the PCAP communication have been specified in [2.8, 2.9].

The Service Specification phase starts from creating a domain model of the system. The relationships between the system level services and their users become candidates for *PSAPs – Provided Service Access Points* of the system level services. The PSAPS are logical interfaces attached to the classes with ports. The domain model for the *Positioning* system and its service *PositionCalculation* is shown in Fig 2.1a and PSAP of the Positioning system is shown in Fig 2.1b.

A valid execution order of signals on PSAP can be specified by the corresponding use case and sequence diagrams. Finally, we formally describe the communication between a system level service and its user(s) in the state machine as illustrated in Figure 2.1c.





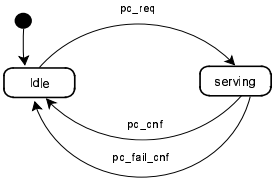
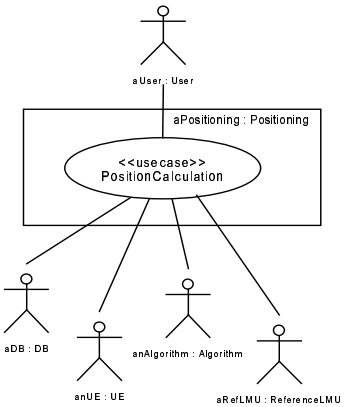


Fig 2.1a. Domain model **Fig 2.1b. PSAP of Positioning System** **Fig 2.1c. State diagram of PSAP communication**

To implement its own services, the system usually uses external entities. The services provided by the external entities partition execution of the *PositionCalculation* service into the corresponding stages.

In the next – *Service Decomposition* – phase we introduce external service providers into the domain model constructed previously, as shown in Fig 2.2a. The model includes the external service providers *DB, UE, LMU* and *Algorithm*. The logical interfaces are attached to the corresponding classes via ports called *USAPs – Used Sevice Access Points* as presented in Fig 2.2b.



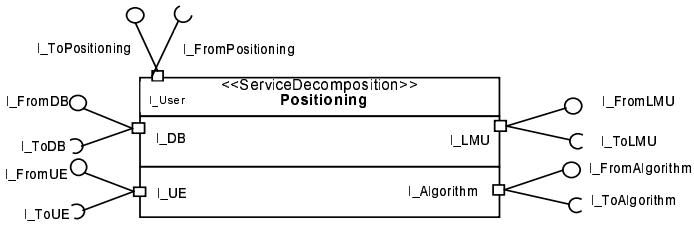
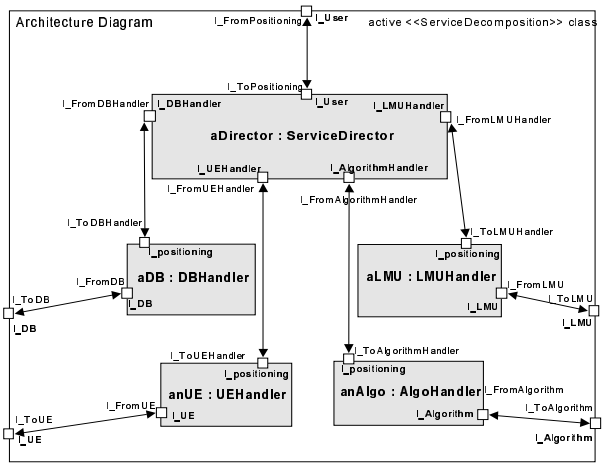


Fig 2.2a. Domain model **Fig 2.2b. PSAP and USAPs of Positioning**



2.2c. Functional architecture

To specify the required stages of service implementation, we decompose the behaviour of the main use cases accordingly. The functional architecture is defined in terms of service components, which encapsulate functionalities related to a single execution stage or other logical piece of functionality. In Fig 2.2c we present the architecture diagram of the *Positioning* system. The main element, *ServiceDirector,* plays two roles: it manages the execution control in the system and handles the communication on the PSAP.

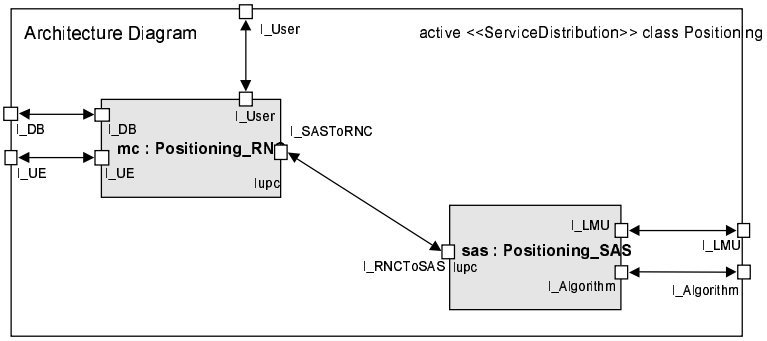


Fig 2.3. Architecture of service distribution

The modular system model produced at the Service Decomposition phase allows us to analyse various distribution models. In the next phase – Service Distribution – the service components are distributed over a given network architecture. Signalling protocols allow for communication between the service components in distant network elements.

In Fig 2.3 we illustrate the physical structure of the distributed positioning system. *Positioning_RND* and *Positioning_SAS* represent network elements in a UMTS network. We map the functional architecture to the physical structure by including the service components into the network elements. The functionality of *ServiceDirector* specified at the Service Decomposition phase is also decomposed and distributed over the given network.

Finally, at the *Service Implementation* phase we specify how the virtual PDU communication between entities in different network nodes is realized using the underlying transport services. The detailed discussion of this stage can be found elsewhere [2.6, 2.8, 2.9].

2.3.2 Formal Service-Oriented Development

The B Method [2.3] is an approach for the industrial development of highly dependable software. The development methodology adopted by B is stepwise refinement. We employ the B Method as a formal framework for verifying the Lyra development process.

In Lyra, the notion of a service component is central to the entire development process. Therefore, to formalise Lyra in the B Method, we have to start by modelling a service component in the B specification language.

A service component has two essential parts: functional and communicational. The functional part is a "mission" of a service component, i.e., the service(s) which it is capable of executing. The communicational part is an interface via which the service component receives requests to execute the service and sends the results of service execution.

Usually execution of a service involves certain computations. We call the B representation (the corresponding data and operations) of this part of service component an *Abstract CAlculating Machine (ACAM)*. The communicational part is correspondingly called *Abstract Communicating Machine (ACM),* and it includes the data and operations modelling the communication channels between a service component and its environment. The entire B model of a service component is called *Abstract Communicating Component (ACC)*.

In Lyra , a service component is usually represented as an active class with the PSAP attached to it via the port. In addition, the state diagram depicts signalling scenario on PSAP including the signals from and to the service consumer. Essentially these diagrams suffice to specify the service component according to the pattern *ACC*. The general principle of translation is shown in Fig 2.4.

The UML2 description of PSAP of service component *Positioning* is translated into the *ACM* part of the B machine ACC_Positioning. The *ACAM* part of ACC_Positioning contains a single operation which abstractly models calculation of an approximate user position. These translations formalise the *Service Specification* phase of Lyra.

In the next phase of Lyra development – *Service Decomposition* – we decompose the positioning service provided by the service component into a number of stages (subservices). The service component can execute certain subservices itself as well as request the external service components to do it. At the *Service Decomposition* phase two major transformations are performed:

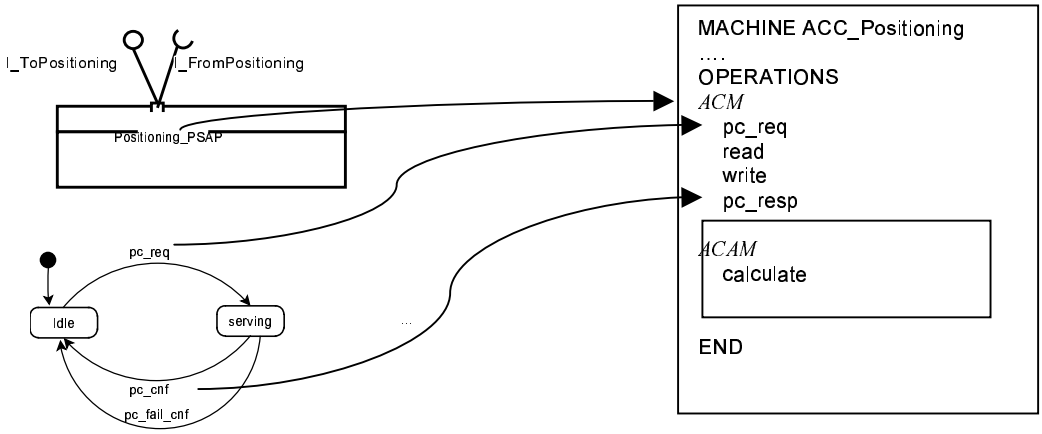


Fig 2.4. Translating UML2 model into the ACC pattern

- the service execution is decomposed into a number of stages (or subservices), and
- communication with the external entities executing these subservices is introduced via USAPs.

Each transformation corresponds to a separate refinement step in our approach.

According to Lyra, the flow of the service execution is orchestrated by *Service Director*. It implements the behaviour of PSAP of the service component as specified earlier, as well as co-ordinates execution by enquiring the required subservices from the external entities according to the defined execution flow.

Service component *Positioning* specified by the machine ACC_Positioning relies on the provided subservices *DB_Enquiry, UE_Enquiry*, *LMU_Measurement*, and *Algorithm_Invocation*. Moreover, the state machine of *Service Director* defines the desired order of execution. In B such decomposition can be represented as a refinement of the abstract model ACC_Positioning.

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

1) the service director of *Positioning* is "centralized", i.e., it resides on a single network element,
2) the service director of *Positioning* is "distributed", i.e., different parts of execution flow are orchestrated by distinct service directors residing on different network elements. The service directors communicate with each other while passing the control over the corresponding parts of the flow.

In both cases the model of service component Positioning with USAPs looks as shown in Fig. 2.2b.

In the first case, it is easy to observe that service component *Positioning* plays a role of the service consumer for the corresponding service components *DB, UE, LMU* and *ALG*. We specify these service components as separate machines according to the proposed pattern *ACC*. The process of translating their UML2 models into B is similar to specifying *Positioning* at the *Service Specification* phase.

Besides defining separate machines to model external service components, in this refinement step we also define the mechanisms describing the PSAP-USAP communication between them (see Fig.2.5).

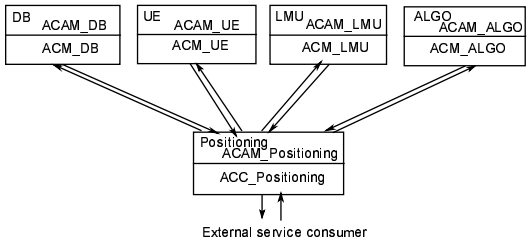


Fig 2.5. Architecture of formal specification (case 1)

Modelling the case of the distributed service director is more complex. Assume that the execution flow of service component *Positioning* is orchestrated by two service directors: the *RNC_ServiceDirector* and *SAS_ServiceDirector*. The architecture diagram depicting the overall arrangement is shown in Fig 2.3.

The service execution proceeds according to the following scenario: via PSAP of *Positioning RNC_ServiceDirector* receives the request to provide the positioning service. After the results of *DB_enquiry* and *UE_Enquiry* are obtained, *RNC_ServiceDirector* requests *SAS_Service Director* to execute the rest of the service and return the result back. Upon receiving the results from, it forwards it to *RNC_ServiceDirector*. Finally, *RNC_Service Director* returns to the service consumer the result of the entire positioning service via PSAP of *Positioning*.

This complex behaviour can be captured in a number of refinement steps. At first, we observe that *SAS_ServiceDirector* co-ordinating execution of *LMU_Measurement* and *Algorithm_Invocation* can be modelled as a “large” service component *LMU-ALG* which provides the services *LMU_Measurement* and *Algorithm_Invocation*. We use this observation in our next refinement step. In our consequent refinement step we focus on decomposition of *LMU-ALG*. The decomposition is performed according to the proposed scheme: by the recursive application of the proposed specification and refinement patterns (see Fig.2.6).

At the consequent refinement steps we focus on particular service components and refine them (in the way described above) until the desired level of granularity is obtained. Once all external service components are in place, we can further decompose their specifications by separating their *ACM* and *ACAM* parts. Such decomposition will allow

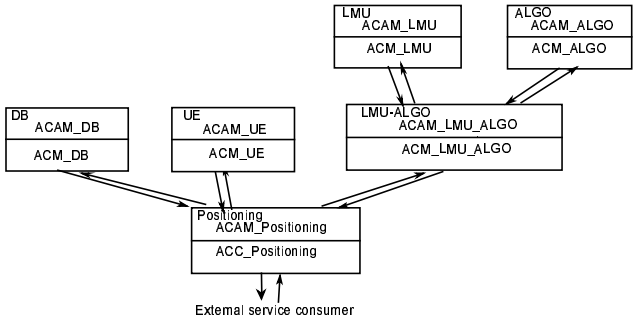


Fig 2.6. Architecture of formal specification (case 2)

us to concentrate on the communicational parts of the respective components and further refine them by introducing details of required concrete communication protocols.

The more detailed description of the proposed approach can be found in [2.4, 2.5].

2.4. 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. prototypes of the tools supporting automatic translation of the Lyra UML2-based development process into specification and refinement process of the B Method,
3. a prototype of the model-based testing plug-in.

2.5. Future Development of the Case Study

The future work on the case study will proceed along the following directions:

- Development of the methodology for reasoning about fault tolerance in the formalised Lyra development process,
- Definition of precise rules for translating the UML2-based Lyra development into B,
- Integration of model-checking into refinement process to address modelling of network protocol-specific communication,
- Modelling of parallelism in the formalised Lyra development,
- Enhancing the model-based testing methodology in Lyra together with the work on the model-based testing plug-in,
- Automatic translation of the Lyra UML2 models into B using the U2B tool.

The methodology for reasoning about fault tolerance will be developed by integrating more realistic fault tolerance mechanisms (e.g., different types of fault recovery procedures) used in the practice of telecommunication systems.

We have already developed general guidelines for translating the UML2-based Lyra models into B specifications. To make automatic translation of the Lyra models possible,

we are going to turn these guidelines into precise correspondencies between elements of UML2 domain-specific models and the corresponding B specifications.

We are planning to further develop the specification and refinement patterns modelling parallel execution of communicating service components. We envisage that formal modelling of distribution of parallel behaviour over the given network architecture will be an especially challenging problem.

In the created specification pattern of a communicating component we have deliberately specified the communicational part very abstractly. This allows us to instantiate it with virtually any communication protocol. To check dynamic properties as well as reason about fault tolerance and parallelism in protocol communication, we will investigate application of model checking techniques.

Finally, while developing the model-based testing plug-in, we will explore how to use the developed B models of communicating systems as an additional source that can be used for test generation.

2.6. References

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

2.2. RODIN Deliverable D4 – Traceable Requirements Document for Case Studies, Project IST-5111599, February 2005.

2.3. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

2.4. L.Laibinis, E.Troubitsyna, S.Leppänen, J.Lilius, and Q.Malik. Formal Service-Oriented Development of Fault Tolerant Communicating Systems. In Proc. REFT'2005 – *Workshop on Rigorous Engineering of Fault Tolerant Systems*, Newcastle upon Tyne, Technical Report of University of Newcastle, UK, July 2005.

2.5. L.Laibinis, E.Troubitsyna, S.Leppänen, J.Lilius, and Q.Malik. Formal Model-Driven Development of Communicating Systems. TUCS Technical Report, 2005. http://www.tucs.fi/research/series/serie.php?type=techreport&year=2005

2.6. S.Leppänen, M.Turunen, and I.Oliver. *Application Driven Methodology for Development of Communicating Systems*. FDL'04, Forum on Specification and Design Languages. Lille, France, September 2004.

2.7. C.Snook and M.Butler. *U2B – A tool for translating UML-B models into B*, in Mermet, J., Eds. *UML-B Specification for Proven Embedded Systems Design*, chapter 6. Springer, 2004.

2.8. 3GPP. Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. See http://www.3gpp.org/ftp/Specs/html-info/25305.htm

2.9. 3GPP. Technical specification 25.453: UTRAN Iupc interface positioning calculation application part (pcap) signalling. See http://www.3gpp.org/ftp/Specs/html-info/25453.htm

SECTION 3. INITIAL REPORT ON CASE STUDY DEVELOPMENTS FOR CASE STUDY 2: ENGINE FAILURE MANAGEMENT SYSTEM

3.1. Introduction

This section of the D8 report summarises the developments in AT Engine Controls (ATEC)[α] case study "Engine Failure Management System" as part of the RODIN project.

It also discusses how the development of the case study will provide some tangible demonstration that can be used to evaluate the impact of RODIN on Engine Failure Management.

The work on the case study has addressed the following tasks from the Description of Work [3.10].

T1.2.1 Define case study, evaluation plan, measurements and assessment criteria
T1.2.2 Produce an informal specification of a typical engine failure management system
T1.2.3 Use the informal specification to produce a visual formal specification

Contributions have been made to other tasks and are commented on later.

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

Initial RODIN presentation (University of Newcastle September 2004)
This outlined issues on the case study which are to be addressed by RODIN. This included a feasibility study into abstraction of failure management using UML-B.
From this presentation work continued in developing a natural language description of the case study to prepare for the next workshop.

Requirements workshop (Chilworth December 2004)
It provided an introduction to learning the methodology of "Event B" for modelling and a methodology to specify a traceable requirements specification presented by J- R Abrial by means of an example structure requirements document [3.12]
This led us into work into requirement research and its development on the Traceable Requirement Specification.

[α] "AT Engine Controls" was formerly called "VT Engine Controls" (VTEC) which was the name referenced in previous RODIN deliverables.

Presentation on work (Helsinki workshop, March 2005)

A summary of the case study development was presented including some early work on requirement engineering and model development.

The following deliverables and papers have been generated from the Case Study work so far.

RODIN deliverables of case study

D2 (D1.1) Definition and Evaluation Plan [3.6] November 2004
D4 (D1.2) Traceable Requirements Specification [3.7] February 2005
D8 (D1.3) Interim report (this report)

Papers

1. "Rigorous development of reusable domain specific components for complex applications." [3.1]
   (This paper provided exploratory modelling of the domain).

2. "The engineering of generic requirements for failure management" [3.2]

3. "Towards a methodology for rigorous development of generic requirement patterns"[3.3]

3.1.1. Case Study Development

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.
The context of the subsystem is shown in Fig 3.1 below. It illustrates that all the sensor inputs to the control subsystem are handled by the Failure Management Subsystem. The subsystem processes these inputs to provide a managed input to the Control Subsystem which in turn drives the engine.

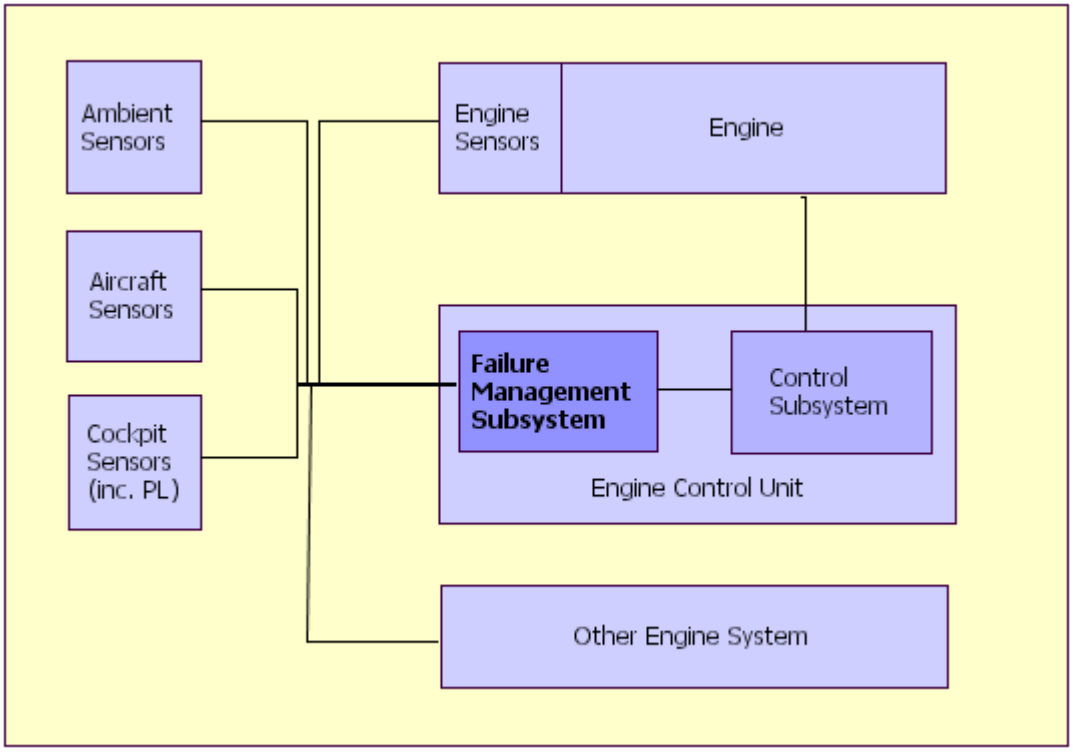


Figure 3.1 – Environment of Failure Management Subsystem

Principally ATECs' aim is to improve an Engine Failure Management Systems' maintenance and re-use, by adopting RODIN methods. The use of the technology is expected to contribute to improvement by

1. Being able to accurately model the domain in order to reduce the semantic gap between application requirement and system design.

2. Promoting re-usability by being able to develop a configurable generic specification.

The application domain is safety critical which makes RODIN rigorous methods a particularly attractive solution for AT Engine Controls.

AT Engine Controls intention is that the development of these models will form the basis towards design and implementation of a future system.

AT Engine Controls have been working closely with the University of Southampton who have provided the necessary support and education in developing a solution using a UML_B approach [3.4].

3.1.2. Case Study Development Cycle

The intention was to develop the modelling into two phases. The development phases are described in deliverable D2 [3.7].

The first phase was to establish the modelling of a representative system using RODIN methods (aim 1). The second phase was to consider more generic functional and component requirements in order to provide re-configurability and reuse (aim 2).

However during the course of development the boundaries of these phases have become less distinct. The creation of a typical requirement specification has resulted in a generic specification. Furthermore the modelling of this general system has identified common components within the system appropriate to generic modelling. In essence some of the work intended for Phase 2 is being addressed in Phase 1. The current intention is to complete the existing model for this phase and then evaluate its generic aspects before entering into the next phase. The generic model is intended to be converted to the new version of the UML_B toolset in Phase 2.

3.1.3. Informal Requirements Stage

A representative requirement of an engine failure management subsystem was produced in natural language format.

The approach taken was for a domain expert to identify the functional requirements common to different engine failure subsystems in order to form a representative engine failure system specification. Some anticipation of future requirements were also incorporated. This resulted in a specification for a general engine failure management subsystem.

The specification described the general functionality supported by detailed instances which were held in tabular form. It supports the idea of a table being used to validate a variant in a generic model. This contributed toward the development of a more generic specification which was developed further in the traceable requirements specification.

3.1.4. Traceable Requirements Specification

The specification is described in detail in deliverable D4 [3.7]. It was generated adopting the format guidance presented in the internal RODIN *Requirements workshop Chilworth* (referenced in the introduction). Its salient features are:-

- A more rigorous natural language definition of generic requirement with traceable reference to the instance data. (This was supported by explanatory text).

- A more rigorous database style definition of the tabular data.

- A taxonomy which identifies the requirement entities of the domain.

- A first cut diagrammatic entity relation model which defines the relationships between these entities.

The approach adopted from the workshop worked well in presenting the requirement, the adoption of the taxonomy was particular suited to labelling common components appropriate to generic design. However it was felt that more research was required in support of developing generic requirements. This led to an investigation into research into domain analysis and domain engineering.

The investigation identified the concept of product line engineering and how it might be applicable to the failure management domain.
The Production Line concept [3.2] nurtures the idea that a template can be used to generate a family of variants of failure detection management systems, this is suited towards developing a generic requirements model for reuse.

Some work on domain analysis and engineering is outlined below and described in more detail in Snook et al [3.2]. It supports ATECs' aim of reusability of an Engine Failure Management System.

3.1.5. Domain Analysis

A core set of requirements were identified from the representative failure management engine system. For example, the identification of magnitude tests with variable limits and associated conditions established several magnitude test types. These types have been further subsumed into a general detection type. This type structure provided the taxonomy for classification of the requirements.

Domain analysis showed that failure management systems are also characterised by a high degree of fairly simple similar units made complex by a large number of minor variations and interdependencies. The domain presents opportunities for a high degree of reuse within a single product as well as between products. For example, a magnitude test is usually required in a number of instances in a particular system. The domain contains a few simple units which are reused many times. A particular configuration depends on the relationships between the instances of these simple units. A first-cut entity relationship model was constructed from the units identified during this stage. The entities identified during domain analysis were:

- **INP** *Identification of an input to be tested.*
- **COND** *Condition under which a test is performed or an action is taken. (A predicate based on the values and/or failure states of other inputs).*
- **DET** *Detection of a failure state. A predicate that compares the value of an expression to be tested against a limit value.*
- **CONF** *Confirmation of a failure state. An iterative algorithm performed for each invocation of a detection, used to establish whether a detected failure state is genuine or transitory.*

- **ACT** *Action taken either normally or in response to a failure, possibly subject to a condition. Assigns the value of an expression, which may involve inputs and/or other output values, to an output.*
- **OUT** *Identification of an output to be used by an action.*

Domain analysis also described the concept of requirement rationale. Considering the rationale behind a requirement is useful in reasoning about requirements in the domain. For example, the rationale for confirming a failure before taking action is that the system should not be susceptible to spurious interference on its inputs. From the consideration of requirements rationale, key issues were identified which served as higher level properties required of the system. An example of such a property would be that the failure management system must not be held in a transient action state indefinitely. The rationale from which it has been derived is that a transient state is temporary and actions associated with this state may only be valid for a limited time. It is felt that the consideration of such rationale and key issues will form an abstract model which is refined by the generic requirement model.

3.1.6. Domain Engineering

The aim of the domain engineering stage is to explore, develop and validate the first-cut generic model of the requirements into a *validated* generic model. At this stage this is essentially an entity relationship model, omitting any dynamic features (except temporary ones added for validation purposes).The UML_B approach adopted for the case study and the supporting tools are described and referenced in the next section 3.2.

The first-cut model from the domain analysis stage was converted to the UML-B notation by adding stereotypes and UML-B clauses (tagged values) as defined in the UML-B profile [3.11]. This allows the model to be converted into the B notation where validation and verification tools are available. The model contains *invariant* properties, which constrain the associations, and ensures that every instance is a member of its class. To validate the model we needed to be able to build up the instances it holds in steps. For this stage a constructor was added to each class so that the model could be populated with instances. The constructor was defined to set any associations belonging to that class according to values supplied as parameters.

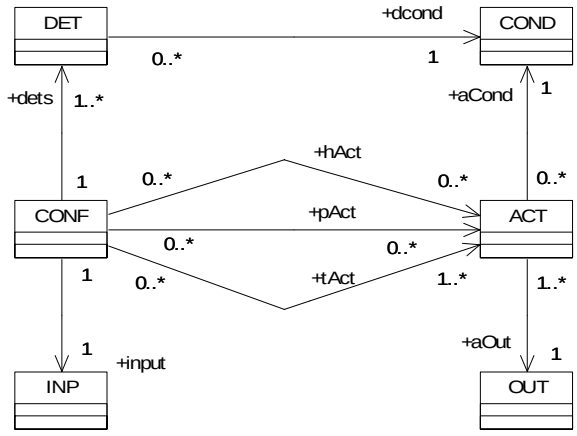


Fig. 3.2. Final UML-B version of generic model of requirements

The model was tested by adding example instances using the animation facility of the ProB model checker (see section 3.2) and examining the values of the B variables representing the classes and associations in the model to see that they developed as expected. The model was re-arranged substantially during this phase as the animation revealed problems. Once we were satisfied that the model was suitable, we removed the constructor operations to simplify the corresponding B model for the next stage. The final version of the UML_B model described above is illustrated in fig 3.2. A detailed description is provided in the generic requirement paper [3.2].

3.1.7. Requirements for a Specific Application

Having arrived at a useful model we then use it to specify the requirements for a particular application by populating it with class instances. We use the verification tool ProB to check the application is consistent with the properties expressed in the generic model. This verification is a similar process to the previous validation but the focus is on possible errors in the instantiation rather than in the model. Although our example is small compared to a real application, there is still a substantial amount of data entry involved and errors were expected. The technique was found to be highly effective, detecting all the data entry errors and satisfying the invariant within a few iterations.

3.1.8. Future Development

The next stage is to add behaviour to the generic model by giving the classes operations. In future work we will investigate the best way to introduce this behaviour during the process. It may be possible to add the behaviour after the static model has been validated as described above. Alternatively, perhaps the behaviour will affect the static structure and should be added earlier. In either case, we aim to formalise the rationale described in the domain analysis and derive the behaviour as a refinement from this.

3.2. Directions on RODIN Methodology and Tools

In order to meet the RODIN objectives the case study is seen to be a driver on RODIN methodology and tools. The following outlines the contribution of the case study to these areas. The next section outlines the experiences with the technology.

3.2.1. Methodology

The case study is intended to drive the methodology by presenting issues and experiences from the problem domain. The adopted methodological approach being used by the case study is modelling using a UML-B approach [3.11].

The experience of developing a generic model described in Snook et al [3.1, 3.2, 3.3] is likely to be the main consideration which will contribute to the RODIN methodology. The impact on methodology is to be described in the methodology deliverable D9 [3.8]. Some experiences using this methodology are outlined in the next section.

3.2.2. Plug-in Tools

The case study is intended to drive the development of plug-in tools by providing some feedback of its experience of using them when applying the methodology. This case study uses plug-ins which support the UML-B approach. However the new Plug-in tools intended for this approach have not been made available at this point (in development for the eclipse platform) this has forced the case study to work with some older existing tools that were available. The tools used by the case study in the UML_B formal development are U2B and ProB.

The UML-B [3.11] is a profile of UML that defines a formal modelling notation. UML-B consists of class diagrams with attached statecharts, and an integrated constraint and action language based on the B AMN notation. It is suitable for translation into, the B language.

The U2B [3.5] translator converts UML-B models into B components (abstract machines and their refinements), thus enabling B verification and validation technology to be exploited. U2B has been implemented using Rationale Rose s' extensibility features this is expected to change to a different implementation in future tool development.

Experience with this version of the UML_B profile may identify areas of improvement in terms of representation ability and ease of use which can be applied to the new tool development.

ProB is a model checker [3.2] and has been used to provide verification and through validation via animation of models developed during the case study.

Verification of the requirement serves to strengthen the mapping of the model to the requirement specification. Verification provides confidence that the UML_B model represents the requirement accurately. It is the closeness of mapping in conjunction with the understandability of the model which will help to reduce the semantic gap (aim1). However the validation of the requirement is particular useful for AT Engine Controls as this allows early exploration of the requirements which can identify unintended behaviour or incomplete requirements which can help improve the dependability of a system and potentially and development cost savings.

The case study is expected to contribute to development of the U2B and ProB tools in terms of how effectiveness and its ease of use for the novice.

Finally in addition to the profile and tools described above it is anticipated that development of the case study will identify new areas of tool support. Some areas have already been identified in the following section.

3.3. Results and Experiences of the Technology

Whilst it is still too early in the project to provide significant evidence to support evaluation, the experience of using the technology so far is summarised below.

3.3.1. Modelling

The stage of development that the model has reached has been described above. The progress is considered to be good by ATEC as the static model developed is already starting to addresses their aims in the case study.

The semantic gap is being reduced as illustrated by the closeness of mapping between the requirements specification and the model components. This was also verified with an instance model.

Reusability is being addressed as the static structure has identified generic components which have been verified against the instance model. However the model still lacks behaviour and scaleability is yet to be tested.

Furthermore exploratory modelling of behaviour has already identified weaknesses in the requirement. Snook et al [3.1] identified that a state can exist where a common transient action such as a freeze can be maintained for a longer period if the individual inputs that share this action fail after each other. Similarly if a given test input is failing erratically then its transient action can persist.

These results provided an early indication that a rigorous approach provides quality benefits.

3.3.2. Learning the Technology

The learning of the RODIN technology for AT Engine Controls has been addressed through the RODIN workshops and working with the University of Southampton.
The following specific areas have been targeted.

UML_B method
UML_B tools
Formal methods Modelling

The learning and applying of a developing technology alongside research activities has presented particular challenges to the novice learner which are summarised below.

1. Mastering the concept of modelling rather than implementation
2. New toolset availability
3. Existing tool functionality limited eg lack of bi-direction relations, unclear error messages
4. Limited UML_B examples to learn from.
5. A comprehensive understanding of formal notation is required to support the UML_B method. ie how to express behavioural concepts in its notation
6. Time/resource available to learning to be shared with research activity
7. Establishing a foundation knowledge in formal methods alongside the UML_B method.
8. Developing the case study model in parallel with learning technology concepts.

Model development of the case study has also identified weaknesses in the existing tools which has identified areas potential areas for new tool development.

3.3.3. Experiences with the tools

UML_B profile
The following improvements were identified using the current profile implementation.

- Ability to make more information visible on diagrams - this will assist in making the model more readily understandable.
- Bi directional associations should be better supported (in the current U2B tool they are translated but navigation in uB is sometimes difficult depending on multiplicities – this should increase both understanding and efficiency in generating models.
- Some restriction to prevent only correct UML_B constructs from being specified would be useful. It was found that incorrect code could be entered which could cause problems when running the ProB tool.

U2B tool
This tool is easy to use. One problem was that the conversion into B was not always successful eg the $ operator being ignored. This bug was corrected in U2B.

Pro B
The tool was successfully used to animate and model check the model.
ProB provides an indicator to show when the invariant is violated. Due to the 'required' (i.e. multiplicity greater than 0) constraints in our generic model, the only way to populate it without violating the invariant would be to add instances of several classes simultaneously. However, we found that observing the invariant violations was a useful part of the feedback during validation of the model. Knowing that the model recognises inconsistent states, is just as important as knowing that it accepts consistent ones.

We found that the analyse invariant facility provided useful indication of where the invariant was violated (i.e. which conjunct) but, in a data intensive model, it is still not easy to see which part of the data is at fault. This is another area for tool improvement.

3.3.4. New Tool Support

Experience of modelling has shown that the size of a failure management requirement can create practical difficulties with the amount of data that has to be input. This became evident when only a few instances were tried when populating the static structure of the model. A requirements manager plug-in tool is envisaged to assist with this problem.

3.4. Demonstrators and Evaluation

The evaluation plan D2 [3.6] has identified three parallel purposes of the case study. These are summarised as

a) to evaluate the UML_B method from AT Engine Controls viewpoint
b) to improve the maintainability and portability of failure management software
c) to provide feedback to the developers of UML-B.

The evaluation of these have been defined by goals and metrics described in the plan. The application of these goals and metrics are ongoing throughout the project and are expected to come to completion in the final report.

The interim reporting of metrics will be limited to what can be demonstrated at particular points.

In phase 1 the intention is to demonstrate a verified and validated visual model of the failure management system.

In phase 2 the intention is to demonstrate a generic system from which variants can be derived and techniques explored and the development of the model to a package explored.

This report has provided some qualitative feedback of the progress so far. An attempt at ranking the evaluation criteria at this point has been given in the deliverable D14 [3.9].

The metrics being collected are

- time spent learning the technology
- problems identified in learning (see above section)
- defects being found in the model/requirements (verification and validation)
- time spent developing models
- improvements identified in tools

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
Accepted for 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, Sept 2005

[3.9] RODIN deliverable D14 : Assessment report 1 IST-5111599, Sept 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"*, Technical Report, Electronics and Computer Science, University of Southampton.

[3.12] J-R Abrial *Mechanical Press*: Requirement Document Nov 2004

SECTION 4. REPORT ON THE CASE STUDY: FORMAL TECHNIQUES IN MDA CONTEXT

4.1 Case Study Description

This case study is primarily aimed at constructing and verifying platforms rather than aiming at the applications themselves. We take the view that with the increased use of technologies such as MDA - which emphasise the role of domain models of applications which are independent of many implementation technologies - the inclusion of properties such as fault tolerance can be decided and architected upon at a later stage of development in much the same way as a choice of implementation platform or language can be made [4.18].

It is also commonly true that platforms are being constructed in parallel with the applications that will run upon those architectures; whether this is a good situation or not depends upon the processes and management involved. Most MDA literature suggests that there is a single platform and that a PIM is mapped (architected) onto this platform to produced the PSM. We take a more liberal view which states that platforms themselves are collections of properties [2] - the theory of this is still under construction in this particular context and will be expanded upon during this course of this case study.

To fulfil this we aim in this case study to take a more aspect oriented approach to the notion of a platform by weaving together a successive series of platforms each introducing an addition level of detail. Each level of detail (again, the theory about what a level of detail or "abstraction level" is depends upon the defition of what a platform is and how this is phrased in MDA terminology - we make an attempt in this case study to answer this question). At least within the context of RODIN we shall consider fault tolerance to be a platform in its own right; given our specification of the Nokia NOTA[1] platform we will utilise existing transformations to generate implementation of this, e.g.: C++ (Symbian), Java, SystemC and so on. In addition we shall also produce versions of fault-tolerant version of both the NOTA platform and NOTA services/applications by mapping the specifications onto the "fault-tolerant" platform and then to our existing transformations.

A number of particular questions need to be answered:

- How much information about fault tolerance can be axiomsed in this way? A discussion about the possibilities and limitations is given in [1].
- Does the ordering of transformations make any difference to the overall properties of the system; on particular concern here is that transformations may either over-constrain a model or not be applicable due to variations in the abstraction level of the domain model for transformation?
- Given this platform and aspect based approach, how does this affect the placement of features such as fault-tolerance into the various models being created and how does this affect the transformation or weaving process to compose two or more models?

1. Network-on-Terminal Architecture. This is described later in more detail.

- How does the effect of using an aspect oriented design flow and composition of model affect refinement?

4.1.1 MITA and NOTA

This case study was originally described as MITA (Mobile Internet Technical Architecture). MITA is a large, "grand plan" for the overall structure of the mobile internet. NOTA on the other-hand is one particular implementation of a part of MITA [4.13][4.14][4.15].

NOTA provides us with a more concrete set of requirements but still conforms to the platform based, distributed, service oriented nature that was put forward in the original case study. As far as the RODIN work is concerned NOTA can be considered a refinement or subset of MITA. No changes to the goals of this work have been made.

Additionally we must also consider situations where the services do not follow the rule of being stateless. Combinations of services and applications may themselves deadlock and fail, but the platform must never.

Many defintions of the term service have been proposed. We follow the defintion that a service is a group of publicly available, common functionalities [4.2].

4.1.2 Network-On-Terminal Architecture

NOTA (Network-On-Terminal Architecture) is a platform for mobile devices for service oriented systems [4.2][1]. The platform itself consists of a number of interconnect nodes which provide access points for services and applications to communicate. Via these interconnect nodes, a service can register itself so that its features and functionality are globally (we shall return to clarify this point) available to other services and applications in the system.

NOTA is envisaged as a platform onto which services and applications can be implemented (it provides services registration, discovery and connection facilities). Services and applications therefore must contain (when made NOTA specific) a certain set of functionalities to remain well-behaved within a NOTA system. Ideally services and applications would be formally specified and their compliance formally proven; this might not always be the case [4.24] and the system should be able to recover from mis-behaving services and applications.

4.2 Major Directions in Case Study Development

This first year has seen a change in the application of the RODIN ideas and technologies to a more concrete implementation. However this has given us time to concentrate more on methodological and theoretical aspects of the ideas being investigated in this case study. The major point here is that technologies such as the OMG's MDA [4.16] are loosely defined and the ideas being suggested lack a rigorous theoretical basis.

Regarding the actual practical part of the case study we have

1. Also compare NOTA with WSML (Web Services Management Layer) which provides a similar kind of functionality but not in the embedded scales of terminals we are considering here. http://ssel.vub.ac.be/wsml/

- Investigated the use of B in distributed systems (via MITA ideas)
- Constructed specifications of the NOTA Platform
- Constructed specifications of potential applications and services that might be based on the NOTA platform
- Taken initial steps towards understanding how OO/AOP/MDA can be utilised in the context of RODIN.
- Taken initial steps towards understanding the role of MDA and formal methods in the construction of Service Oriented systems.
- Started investigation of refinement and retrenchment properties between models.

4.3 Achieved Results

4.3.1 Platform Development

Development of platforms is conceptually no different from developing applications themselves; there are of course differences in practise such that platforms are not normally concerned with execution and concentrate more on structure and non-functional properties. In this case study we are more focused on the specification of the structure of the platform, its internal consistency and how non-function aspects can be specified and validated/verified inside the Rodin framework [4.6][4.7]. There exists a large amount of material related to the development of distributed system in a formal manner, e.g.: [4.9][4.12][4.22][4.27][4.25][4.27]. We are therefore particularly interested in the combination of these formal techniques with MDA and its ideas of model composition through transformations.

We consider two aspects of platform development here, the first is related to the MDA development flow and the role of verification/validation within that flow and the second is related to development methods used within Nokia.

The case study is being designed to evaluate the following scenario:

- The platform contains information about where fault tolerance lies and how this is to be achieved.
- Applications are developed independently of this platform. These applications may however contain information about fault tolerance as well as other desired properties.
- These applications may be mapped to many platforms (each having their own structure and properties). In this example we map the application onto NOTA.

The result of this is that we obtain a model of the application now in the context of that platform, i.e.: in MDA terms it is a platform specific model of that application.

The results should then show the following

> "The (platform independent) application should pass all the required tests, i.e.: it verifies and is validated according to whatever criteria have been specified at this level."

and similarly for the platform:

> "The platform specific model created by the composition or mapping of the application onto the platform should satisfy all the relevant criteria respectively."

We do know from experience that the following scenarios do exist:

- The properties specified on the platform independent application can not always be implemented by the chosen platforms.
- The composition of properties results in an overly constrained system which then can not perform the tasks or behave in the way initially envisaged.

The former case is often seen when there exists pollution of properties from other levels of abstraction and platforms into the platform independent model of the application. This sometimes can not be avoided.

In the second case it is necessary to weaken the specification in such a way that the desired behaviour of the application and platform are preserved [4.1]. Weakening specifications means that the refinement relationship breaks and various properties of the system are potentially compromised [4.5]. Techniques such as retrenchment [4.23] however make this process very explicit and thus safer in the sense that the breaking of the refinement relationship is made explicit and can be verified.

4.3.2 NOTA Platform Specification

The interconnect is the platform that is being designed for a new series of mobile devices. This platform is designed to consist of a number of nodes (called interconnects) that are capable of running applications and services. The actual implementation of these is left undefined as potentially an interconnect node might be a hardware, software or mixed component. All interconnect nodes however are capable of communicating by some standard means. A model of the interconnect node and related elements is shown in figure 4.1.

The specification here exists at a different "abstraction" level than the service/application specifications. The models here describe the interconnect platform onto which services and applications are implemented. For example this can be visualised using the Y-model or "standard" MDA approach. Compare this with, for example, the SPARC processor/architecture definition from Sun.

Regarding the sockets, an interconnect node keeps track of the sockets that have been created:

Local Socket

> This is a socket between services on the same interconnect node

Remote Socket

> This is a socket between a service on the current interconnect node and some other interconnect node.

Target Sockets

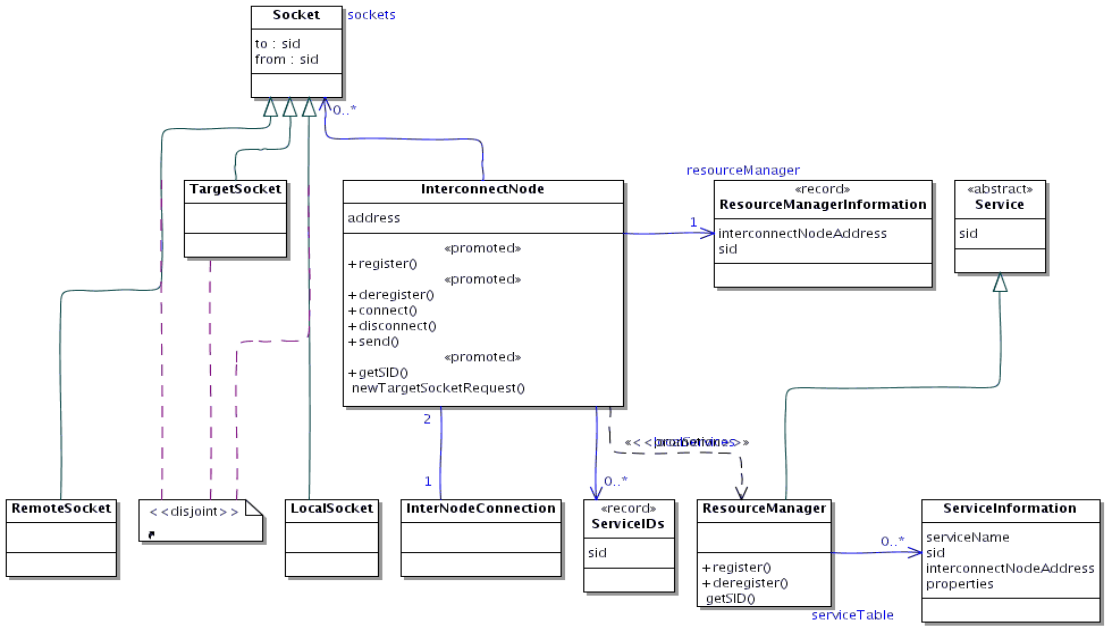


FIGURE 4.1. Class Diagram of the Interconnect Platform

These are sockets which have been requested by other interconnect nodes.

The typical boot sequence is described in the sequence diagram in figure 4.2 on page 6. Note the artistic license in this sequence diagram, this is due to so vagaries of a so called “UML compliant” tool. This diagram shows two cases, namely the boot sequence which creates the interconnect, its connections and a resource manager; of which there is one in any system. We also describe the starting of a service which registers itself with one of the nodes. For simplicity the creation of the sockets to be used during registration is not shown on this diagram for brevity.

Before describing the sequence of events for registration it is important to describe informally the workings of each of the operations described in the interconnect node class diagram.

InterconnectNode::register

When a service registers, it supplies its service name and properties. These properties include information about what API the service provides and other information, for example, security keys etc. This information is forwarded to the resource manager for processing; it is the resource manager that has the responsibility for deciding whether a service is allowed to register and issuing service IDs.

As a return either a service ID and acknowledge is returned or a null value and an error message (i.e.: RMERROR).

During this process the local Services table must be updated to contain this new service ID.

InterconnectNode::deregister

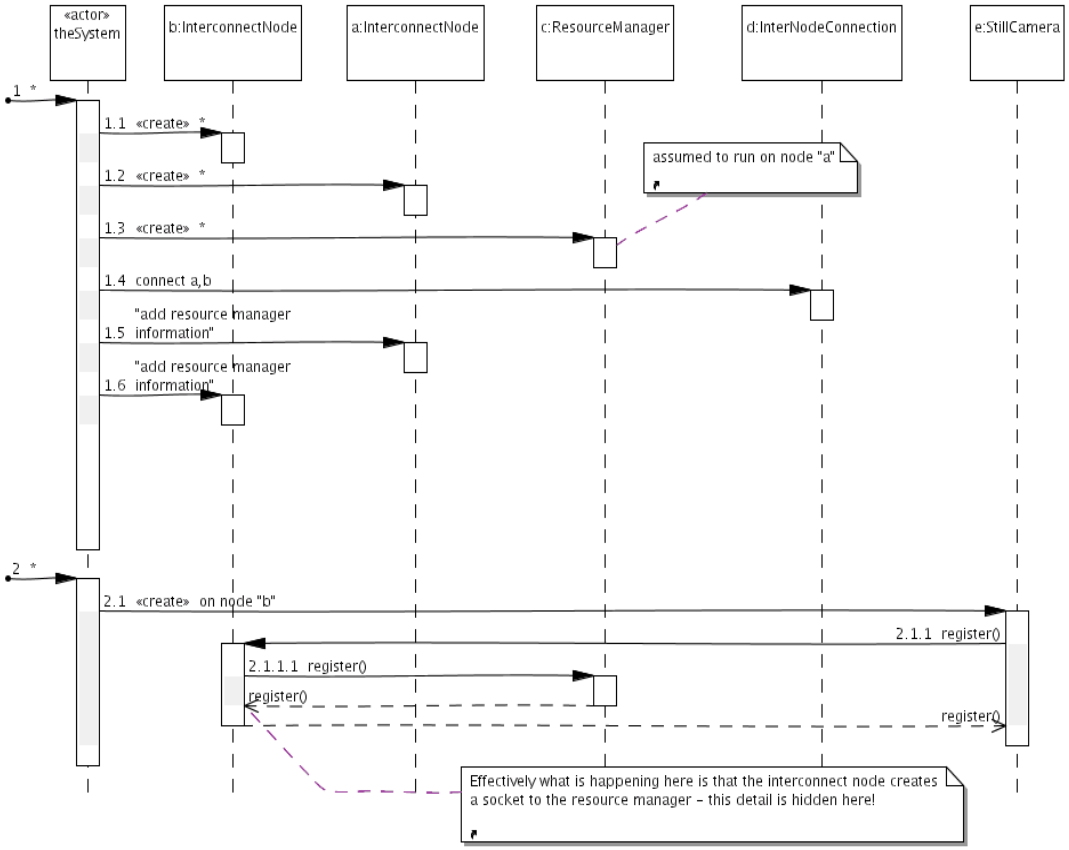


FIGURE 4.2. Example Interconnect Boot Sequence

A service announces that it is no longer functioning (termination, fault or any other reason for suspension of services) and that it is to be removed from the global pool of services. This information is forwarded to the resource manager that decides upon the course of action. Only services themselves may deregister, a service can not be deregistered by another service.

If the resource manager allows deregistration then an acknowledgement is sent back and the service is also removed from the locals services IDs for that interconnect node. It is also possible that a refusal is made an no changes may be made to the local services. It is possible in this situation that the service may still be available but refuse all attempts to access it afterwards.

InterconnectNode::connect

A service will connect to another service by its service ID. The sid if local will result in the creation of a socket to the given service and INSUCCESS is returned along with the id of the socket.

If the sid is not in the local Service table then a connection is made to the resource manager which supplies information about the location of the service and whether that service id exists. The resource manager will either respond with RMSUCCESS denoting that the serv-

ices does exist; then a remote socket will be created and INSUCCESS returned along with the socket id to the service.

If the resource manager refused and returns RMERROR then no socket is created and INERROR is returned.

InterconnectNode::disconnect

> The id of a socket is given. This socket must be owned by the requesting sid in which case the socket obviously must exist; in all this cases the socket is removed and INSUCCESS is returned, else INERROR.

InterconnectNode::send

> Given a socket id which is owned by the calling service, some data is sent via this connection. At this point in time we assume that this is always successful returned INSUCCESS.

InterconnectNode::getSID

> Given a service name string then this initiates a connection to the resource manager which returns the SID of the requested service. If this service exists then INSUCCESS and the SID are returned, else INERROR.

InterconnectNode::newTargetSocketRequest

> When a connection is made between interconnect nodes by a call to the connect operation, the underlying (or whatever) subsystem makes the "physical" connection. The target interconnect receives a call to make a new socket for the "physical" connection. If this is possible then a new target socket is created, else an error is returned (INERROR). After creation it is necessary to inform the service that this socket is aimed at that a socket is available. If this succeeds then INSUCCESS is returned via this new socket, else INERROR.

Service IDs uniquely identify a service within the local system (i.e.: current device and associated peripherals). Service IDs are defined to be an integer value in the range 0..infinity. However a service ID of 0 is always an error. All operations that accept service ID, if they accept a service ID of 0 then an error signal is returned, e.g.: RMERROR, INERROR.

4.3.2.1. B Translations

Currently the main focus of development is by constructing the structural models and assigning responsibilities in UML and then writing the invariants and operations in B. These together are then translated into a pure B representation and then presented to AtelierB for theorem proving and then additional *validation* through animation and model checking by ProB.

The primary difficulties are at this time:

- keeping the B and UML models in sync. Tool support is lacking although work in ongoing with U2B.
- maintaining consistency across the various projected specifications; B is not object oriented.

The specifications currently (at least at this abstraction level) are relatively simple and the nature of B keeps us focused on the visible interfaces and the more declarative aspects of the modelling process. We have a direct comparison here with ongoing (and now in the light of the results and techniques here highly modified) work in SDL for constructing specifications of the interconnect node. The primary result is that the current SDL work has been abandoned and that any new work is directly based by hand translation from the UML/B to SDL; we are investigating automating this using augmented XML representations of B.

An extract of the specification can be seen below[1] - we focus here mainly on the invariant and a number of the operations:

```
MACHINE notaic
...
INVARIANT
 sids <: SID & externalSids <: sids & sockets <: SOCKET &
 localServices <: sids & address : ADDRESS &
 resourceManager_interconnectNodeAddress <: ADDRESS &
 resourceManager_sid <: sids &
 interconnect_resourceManagerAddress <: ADDRESS &
 interconnect_resourceManager_sid <: sids &
 ( not( resourceManager_sid = {} ) => card(resourceManager_sid) = 1 ) &
 localServices / externalSids / resourceManager_sid = sids &
 localServices / externalSids / resourceManager_sid = {} &
 localSocketsTo : sockets <-> sids &
 remoteSocketsTo :  sockets <-> sids &
 localSocketsFrom : sockets <-> sids &
 remoteSocketsFrom :  sockets <-> sids &
 targetSocketsTo : sockets <-> sids &
 targetSocketsFrom : sockets <-> sids &
 dom(localSocketsTo) / dom(remoteSocketsTo) = {}
...
OPERATIONS

 /* OO KLUDGE! */
 resourceManagerExternalRegistation =
 PRE
  not( resourceManager_sid = {} &
       resourceManager_interconnectNodeAddress = {} )
 THEN
  ANY   newEsid
  WHERE
   newEsid : SID - sids &   not(newEsid : localServices) &
   not(newEsid : resource  WHEREManager_sid )
  THEN
   externalSids := externalSids / { newEsid } ||   sids := sids  / { newEsid }
  END
 END ;
...
rr,ss<--interconnectNode_register(nn) =
  /* nn is the name of the service,
     rr is the return code
     ss is the sid
  */
 PRE
  nn : NAME &
  not( interconnect_resourceManager_sid = {} &
       interconnect_resourceManagerAddress = {} )
 THEN
  CHOICE
   ANY    newsid
   WHERE
   newsid : SID - sids &   not( newsid : externalSids ) &
   not( newsid : resourceManager_sid )
   THEN
   sids := sids / { newsid } ||   ss := newsid ||
   rr := INSUCCESS ||   localServices := localServices / { newsid }
   END
  OR
   ss :: SID ||   rr := INERROR
```

1. "..." denote where the specification has been cut

```
  END
 END ;
...
rr,so<--interconnectNode_connect(cc,ss) =
 /* cc is the calling service,
    ss is the sid that it wishes to call to,
    so is the socket created,
    rr is the return code
 */
 PRE
  cc : sids &  cc : localServices &  ss : sids &
  not(cc = ss) &  not( ss : resourceManager_sid )
 THEN
  CHOICE
   rr := INERROR ||   so :: SOCKET
  OR
   ANY    newso
   WHERE  newso : SOCKET - sockets
   THEN
   sockets := sockets / { newso } ||   so := newso ||   rr := INSUCCESS ||
    IF     ss : localServices
    THEN
      localSocketsTo := localSocketsTo / { newso |-> ss } ||
      localSocketsFrom := localSocketsFrom / { newso |-> cc }
    ELSE
     remoteSocketsTo := remoteSocketsTo / { newso |-> ss } ||
     remoteSocketsFrom := remoteSocketsFrom / { newso |-> cc } ||
     externalSids := externalSids / { ss }
    END
   END
  END
 END  ;
...
END
```

Note the inclusion of (horrors!) such as the first (suitably marked) operation to allow us to simulate activities external to the interconnect node. In this case to generate a number of external services which may be running on other interconnect nodes other than the current. This is one of the problems when trying to project single classes out of a UML model.

However, the results from theorem proving have shown that the internal structures and behaviour of the node remain consistent across the operations - this has greatly simplified and improved the reliability of the SDL implementation.

Results from this specification's processing by AtelierB can be seen below

```
Printing the status of notaic
        notaic AutoProved /home/ioliver/BProjects/NOTAIC/notaic.mch
+-------------------------------------+-------+------+-------+-------+------+-----+
|                                     | NbObv | NbPO | NbPRi | NbPRa | NbUn | %Pr |
+-------------------------------------+-------+------+-------+-------+------+-----+
|   Initialisation                    |     3 |    7 |     0 |     7 |    0 | 100 |
|   resourceManagerExternalRegistation|     9 |   13 |     1 |    12 |    0 | 100 |
|   createResourceManager             |    10 |   14 |     0 |    14 |    0 | 100 |
|   announceResourceManagerLocation   |    18 |    0 |     0 |     0 |    0 | 100 |
|   interconnectNode_register         |    28 |   13 |     1 |    12 |    0 | 100 |
|   interconnectNode_deregister       |    20 |   13 |     1 |    12 |    0 | 100 |
|   interconnectNode_connect          |    45 |   16 |     0 |    16 |    0 | 100 |
|   interconnectNode_disconnect       |    33 |    6 |     2 |     4 |    0 | 100 |
|   getSID                            |    39 |    0 |     0 |     0 |    0 | 100 |
|   newTargetSocketRequest            |    34 |    6 |     0 |     6 |    0 | 100 |
+-------------------------------------+-------+------+-------+-------+------+-----+
| notaic                              |   239 |   88 |     5 |    83 |    0 | 100 |
+-------------------------------------
```

All proof obligations have been discharged - the specification here is relatively simple - in the cases where the interactive theorem prover has been needed the proofs have been discharged by the predicate prover without difficulty.

Proofs that have not been amenienable this way have invariable lead to discoveries of errors in the specification itself (misspecification) or misunderstandings in the requirements.

It must be stressed that on many occasions we have had a theorem proved specification but results from the model checking and animation in ProB have shown that the specification is wrong and that the specification is behaving not according to our wishes.

The specification of the sockets is performed similarly:

```
MACHINE socket

SEES notaGenerics

CONSTANTS capacityI, capacityO

PROPERTIES capacityI : NAT &  capacityI > 0 &  capacityO : NAT &  capacityO > 0

VARIABLES incommingBuffer, outgoingBuffer

INVARIANT
 incommingBuffer : seq(DATA) &  outgoingBuffer : seq(DATA) &
 card(incommingBuffer) <= capacityI & card(outgoingBuffer) <= capacityO

INITIALISATION incommingBuffer := [] || outgoingBuffer := []

OPERATIONS
 /* API calls to a NOTA Process */
 send(dd) =
 PRE  dd : DATA & card(outgoingBuffer) < capacityO
 THEN  outgoingBuffer := outgoingBuffer <- dd
 END;

 dd<--getData =
 PRE  not(incommingBuffer = [])
 THEN  dd := first(incommingBuffer) || incommingBuffer := tail(incommingBuffer)
 END ;

 flushIncommingBuffer =
 BEGIN  incommingBuffer := []
 END ;

 flushOutgoingBuffer =
 BEGIN  outgoingBuffer := []
 END ;

 rr <-- isDataAvailable =
 BEGIN
  IF   incommingBuffer = []
  THEN    rr := FALSE
  ELSE    rr := TRUE
  END
 END ;

 /* Calls to the transport layer .. here for convenience
    to allow simulation of sending the data through a pipe */
 pipeSend =
 PRE  not(outgoingBuffer = [])
 THEN  outgoingBuffer := tail(outgoingBuffer)
 END ;

 pipeReceive =
 PRE  not(card(incommingBuffer)=capacityI)
 THEN
  ANY   dd
  WHERE dd : DATA
  THEN  incommingBuffer := incommingBuffer <- dd
  END
 END
END
```

and similar problems as described earlier have been found here.

Matching the results from the sockets with the results from the interconnect node are problematic. We assume that if it were possible to work in a truly object oriented way then a system with many concurrent sockets managed by many concurrent interconnect nodes (as according to the class diagram in figure 4.1) would be correct. We are currently investigating ways this can be achieved in B without resorting to U2B to generate the specifications.

One interesting facet here has been the model checking of the behaviour of the sockets. ProB is capable of generating graphs of the state space of a machine. We have tools available internally which can also explore these graphs and compare these with logging information returned by implementations. An example of the state space for sockets (with limited SET sizes) can be seen in figure 4.3.

Despite the apparent incomprehensibility of this graph, it is however generated from a representation in the dot language - part of the GraphViz graph visualisation package[1]. Two results come from this:

- The textual format is processable as described earlier and can be compared against log files generated from implementations.
- The visual appearance of the graph gives clues to major branching or other problems such as deadlocks.

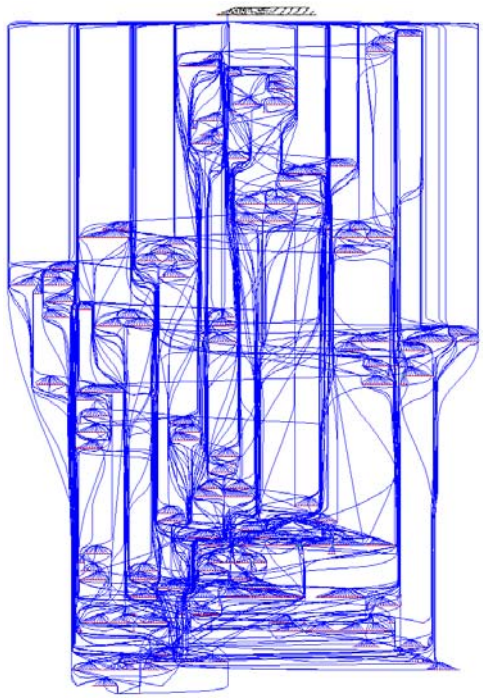


FIGURE 4.3. State Space Graph for Socket.mch

Of course the latter case is purely subjective but generally the "cleaner" the graph looks the better.

4.3.3 Example Service Specifications

We present here two potential services that could be implemented upon NOTA. As we have described earlier, the specification of these services is independent of NOTA and could be implemented on any given platform.

4.3.3.1. Still Camera Service

The basic functionality of a camera service is to make available the API functions for taking pictures (obvious). In this case there are five pieces of functionality that we consider:

- taking a single picture
- taking multiple pictures (aka multishot mode)
- changing the number of pictures to be take in multishot mode
- changing between multishot and single shot modes
- changing between camera states (i.e.: on, standby etc)

1. http://www.graphviz.org/

The UML description for a camera is fairly simple as can be seen in the figure 4.4 which shows the simple class diagram and figure 4.5 which describes the behaviour in terms of an orthoginal state machine.

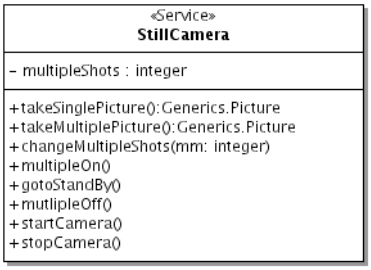


FIGURE 4.4. Class Diagram of Still Camera

The attributes and operations have the following natural language specifications:

multipleShots: this is a value between 2 and 10 which states how many shots should be taken sequentially in multiple shot mode

takeSinglePicture: the camera must be in the on state and multiple shot mode must be off. The result is a single picture returned.

takeMultiplePicture: the camera must be in the on state and multiple shot ode must be on. The result is a set of pictures - the number of which is given by the number of multiple shots to be taken.

changeMultipleShots: this accepts an integer which is used stored in the multiple shots attribute

startCamera: switch the camera to on mode - this can occur at any time

stopCamera: switch the camera off - this can occur at any time

gotoStanby: this places the camera is stand by (aka power save) mode. This mode can only be activated from the on mode. This operation would not normally available to the user.

The camera can be considered at any point in time to be in certain discrete states which can be represented by an orthogonal state machine shown in figure.

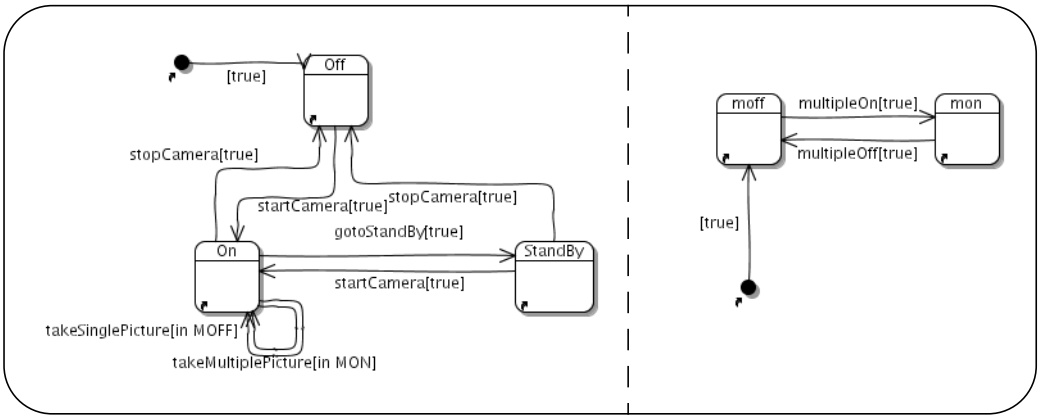


FIGURE 4.5. Still Camera State Machine

What is not shown in this example is the specification of the specifics of the behaviour. This is embedded inside the elements diagram as allowed by the various tools in use.

4.3.3.2. Storage Service

The storage service provides a common API to all storage devices. The basic functionality provided is:

- storing of items (files)
- retrieving of items (files)
- overwriting of items (files)
- deletion of items (files)
- renaming of items (files)

The UML class diagram of the storage is shown in figure 4.6 and as before the natural language specifications of the attributes and operations are given as below.

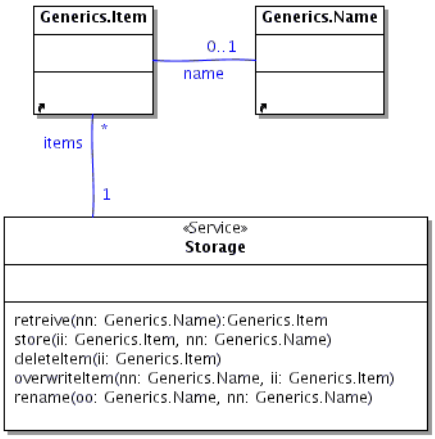


FIGURE 4.6. Class Diagram of Storage

items: the set of items that the storage contains - basically an index or directory. Each item has a unique name

store: takes an item and name and stores it. An item with the given name must not already exist in the storage

retrieve: given an name that exists in the storage, returns the associated item

rename: given a name of an item that exists in the storage, changes its name to the new given name. The new name must not previously exist in the storage

overwriteItem: given a name and item where the name already exists in the storage, overwrite the currently stored item with that name with the given item.

deleteItem: given the name of an item that already exists in the storage, remove that item and the corresponding name.

Similarly the storage specification in B is very simple and is concerned with maintaining the set of items:

```
MACHINE storage

SEES generics

VARIABLES items

INVARIANT items : NAME <-> ITEM

INITIALISATION items := {}

OPERATIONS
 storeItem(ii,nn) =
 PRE
  nn : NAME &  ii : ITEM &  nn /: dom(items)
 THEN  items := items / { nn |-> ii }
 END;

 overwriteItem(ii,nn) =
 PRE  nn : NAME &  ii : ITEM &  nn : dom(items)
 THEN  items := items <+ { nn |-> ii }
 END;

 ii <-- retreiveItem(nn) =
 PRE  nn : dom(items)
 THEN  ii := items(nn)
 END;

 deleteItem(nn) =
 PRE  nn : dom(items)
 THEN  items := { nn } <<| items
 END;

 renameItem(oo,nn) =
 PRE  oo : dom(items) &  nn : NAME &  oo : NAME &  nn /: dom(items)
 THEN  items := items - { oo|->items(oo) } / { nn|->items(oo) }
 END
END
```

Again all proof obligations were discharged without problem.

4.3.3.3. Example Camera Application Specification

We now describe an simple application that utilises the services described above. This application manages the available cameras and storage devices and simple allows the user to choose between these and take photographs. The photograph is then stored by the selected storage service.

There application also keeps track of the last picture taken. This application at this time insists that the camera only takes single shots (i.e.: multishot mode is always off).

The model describing the structure of this application can be seen in figure 4.7 and similarly the natural languages descriptions of its functionality below:

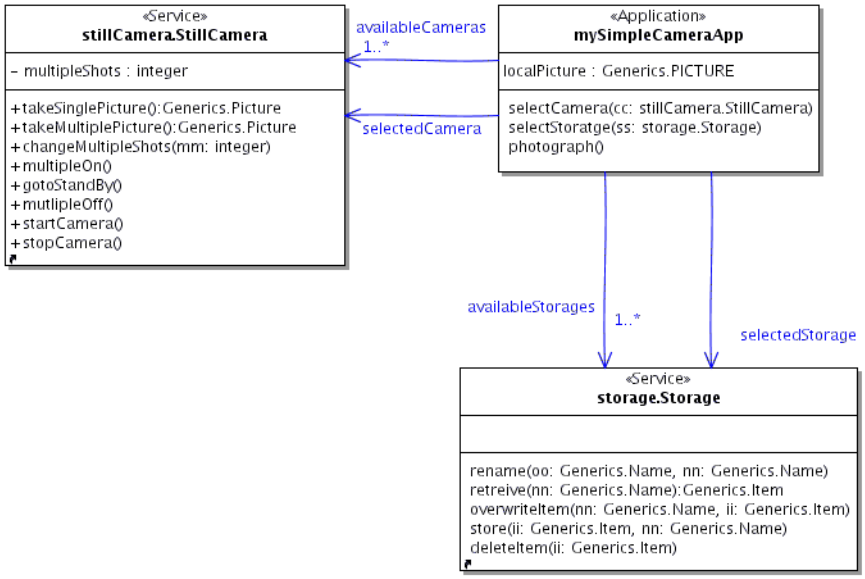


FIGURE 4.7. Class Diagram of Simple Camera Application

The following are natural language specifications of the attributes and operations of the camera application:

localPicture: contains a picture which is the last picture taken by the camera

availableCameras: lists all the available camera (services). Some mobile devices have more than one camera device.

availableStorage: lists all the available storage (services). Some mobile devices have more than one storage device.

selectedCamera: points to the currently selected camera, which must be available to the camera application.

selectedStorage: points to the currently selected storage which must be available to the camera application.

selectCamera: chooses one of the available cameras

selectStorage: chooses one of the available storage services

photograph: take a photograph. The photograph will be stored as the localPicture and be automatically stored by the selected storage service. An example of this in use can be seen below.

The workings of the photograph() operation is described using the interaction diagram in figure 4.8.

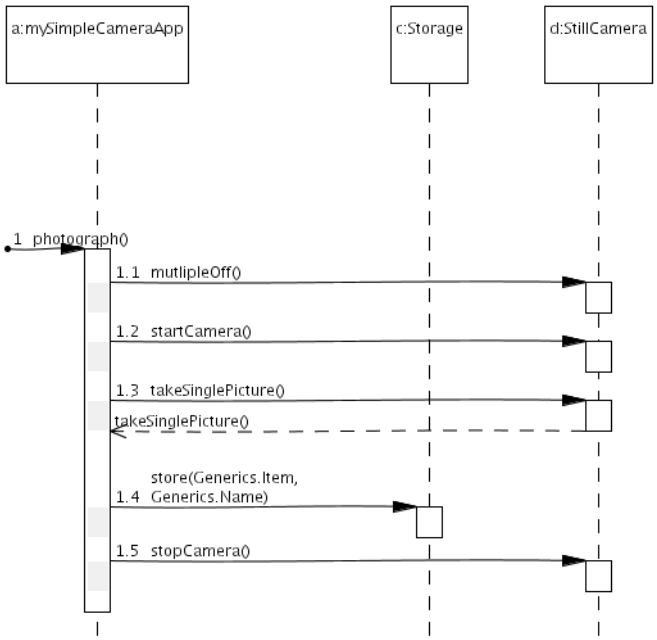


FIGURE 4.8. Interaction Diagram for photograph() Operation

To construct the application we must compose the specifications for storage, camera and the application itself. The first major problem here is that we need individual objects to represent various instances of cameras (e.g.: the Nokia 6680 has two individual cameras), similarly storage, most mobile devices have memory cards, SIM cards and even hard disks; off-device storage is also possible.

4.3.3.4. Commonalities

When working at this "domain" modelling level with the services and applications we have found it necessary to augment the basic UML with a number of stereotypes and also we have identified a number of common generic artifacts.

The stereotypes are simply <<service>> and <<application>>. These are applied to the class which takes the role or responsibility of managing the collaborations in that service or application.

The <<service>> stereotype states that this particular structure will be thought of as a service - that is in the future it may become (platform permitting) some kind of service component. It is envisaged that services in the future will be placed in some kind of respository to form a library of standard services; cf. standard libraries in some programming languages/environments.

Similarly the <<application>> stereotype states that this particular structure will be thought of as an application.

The primary difference here being is that services can be used by anything, while applications only call services. Applications can not be called or used as services.

The common or generic artifacts relate to more the standardisation of a set of data types or classes found at this level. The class diagram in figure shows these currently.

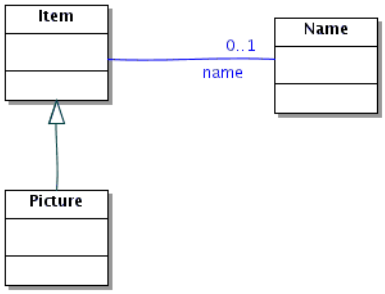


FIGURE 4.9. Domain Level Generic

An item is any piece of data (or in object oriented terms, an object of some kind), a picture is a special kind of object and a name is some way of naming something. Items may or may not have names. Within a given namespace, two items with the same name are necessarily identical. The definition of namespace however is lacking at this point in time other than some informal, "common sense" meaning.

4.3.4 Object Orientation in B

One of the major issues to be tackled in this work is how to combine object orientation with a formal specification language such as B which does not directly support object oriented concepts such as inheritance, polymorphism or the class-object dichotomy.

One issue is the tools that are available [4.4], in previous projects (PUSSEE [4.12]) we have utilised the U2B tool [4.26] to automate the translation from UML to B. Currently in the Rodin project the tools are unavailable; usage of U2B at the moment is difficult due to the differences in tool chain and that most projects now rely upon the UML 2.0 XMI representation which can not be read by U2B at present. While useful, U2B still has limitations such as it does not handle inheritance and that the UML models have their attributes, invariants and operation pre/post conditions written in B - one might argue that B as a constraint language to UML is not a bad thing, but unlike OCL, B is not integrated with the UML meta-model and lacks certain constructs for navigation expressions, typing etc.

The specifications so far discussed in this document have been hand translated either in the style of a U2B translation which embeds the object oriented management constructs into the B in much the same way as early C++ compilers produced convoluted C code to emulate these features directly. The second option has been to individually translate each class as a single B machine construct and attempt to "hide" other classes by presenting these as simple set structures (using B's SETS construct). This latter method is a kind of projection of various structures in the model and while simplifying the specification in B does not allow the whole model to be checked at once.

Neither method is wholly satisfactory at this time but the B produced is amenable to sensible verification and validation, although we do have a number of difficulties proving certain constructs due to the added complexity of the OO management constructs. Particular success has been had using the ProB animation/model checking environment to assist in understanding the relationships (maintenance of relationships between elements of sets representing objects).

We are also currently investigating projecting the inheritance (specialisation/generalisation) structures out of the UML model so that classes can be represented as B machines and the inheritance relationship as refinement relationships between machines. This gives a very strict interpretation of inheritance but allows an alternative way of asserting the Liskov Substitution Principle [4.8]. This method however would not admit multiple inheritance [4.3]; one hypothesises here that the machine inclusion mechanisms might be more appropriate in these situations but one loses the refinement proof obligations.

4.3.5 Theoretical Basis of MDA

We have an outline of a categorical semantics for MDA. This theory provides semantics to the notions of model, platform, language, development flow and the relationships between models: refinement, transformation and translation.

It is hoped that this work will be finalised before the end of this year. The development and application of this theory in being made within the context of this case study and various projects outside of Rodin, e.g.: Åbo Akademi's Coral tool.

4.4 Demonstrators and Evaluation

The evaluation plan described in Rodin Deliverable D1.1 identifies the following criteria for this casestudy; we provide some evaluations from the work performed so far:

- How well does this formal approach fit with existing processes?

  Any formality or rigor in any engineering exercise improves some aspects of the quality of the finalised product. At least the experiences within MITA and NOTA on constructing a specification with the required platform abstraction and formality has lead to results showing potentially gross errors in the existing (non-formal) specification.

- How well do these techniques integrate with an object-oriented approach?

  This still needs more analysis but current results point to (once again) the semantic gap between object-oriented and structured decompositions.

- In what form are these technologies transferred to the Nokia Business Units?

  The ideas and concepts have and are being successfully transferred to various commercial products. This has been primary in the way of thinking that is being employed and the emphasis on clear and precise requirements/specification and platform independent designs. The transfer of verification techniques however is still a problem.

- Can this approach check components against a (very loose) specification?

  Yes, but this still needs more analysis before a clear conclusion can be made on this.

- What is required to understand over constraint models, their causes and effects?

  This has so far only manifested itself either as coding errors in the specification or as a pointer to mixing abstraction levels (i.e.: platform or architectural pollution of a model).

Investigations of this phenomenal during MDA style transformations will be investigated during the next steps of this case study.

- Can this approach deal with requirements volatility?

  Techniques such as retrenchment are known to handle this, though there are methodological and theoretical issues here. The existing specification has been reworked a number of times with respect to addition feature requirements but more substantial changes need to be investigated in more detail.

4.5 Summary

Work on this casestudy is progressing well now that steps have been taken to more concretise the application from a very generic distributed framework such as MITA into NOTA

The results of this work with an existing project within Nokia Research has been extremely beneficial but of course has lead to a number of other issues such as how to capitalise on this early specification and verification/validation work later in the development process.

Preliminary results in terms of time spent working with requirements is showing a 60-70% decrease in time spent developing the system from specification to prototype. However some of this can be explained by the personnel involved at this time, e.g.: experts in SDL/C++ coding, B, verification etc. We would expect to see a 30% increase in productivity (equatable with 30% decrease in development time with a more correct product at delivery) once these techniques are more established.

The work will continue in the following fashion; firstly the various specifications will be composed, i.e.: stillCamera, store and the simple camera application will be made NOTA specific using a transformation (MDA mapping/transformation). Via this we will investigate whether or not the transformations are simply superpositions or whether additional effects are introduced.

Secondly the set of fault tolerant properties will be decided upon and these will be integrated again using transformations, i.e.: fault tolerance being considered a platform. We will investigate fault tolerance as a platform with respect both to services/applications and the NOTA platform itself. Various combinations of fault tolerant services/applications and the NOTA platform will be tried and analysed for their respective properties.

4.6 References

4.1 Thomas Bolusset, Flavio Oquendo (2002). Formal Refinement of Software Architectures Based on Rewriting Logic. RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, January 22, 2002 - Grenoble - IMAG

4.2 Manfred Broy (2004). From Objects to Components to Services - A formal framework for service-oriented architectures. WICSA, 4th Working IEEE/IFIP Conference on Software Architecture. June 2004. Oslo, Norway.

4.3 Luca Cardelli. A Semantics of Multiple Inheritance (1988). Information and Computation 76:2-3, Feb/March 1988, pp 138-164.

4.4 Bernhard Steffen (2004). Major Threat: From Formal Methods without Tools to Tools without Formal Methods. ICECCS 2004:

4.5 Andrea Kerschbaumer (2002). Non-refinement Transformations of Software Architectures.Formal Refinement of Software Architectures Based on Rewriting Logic. RCS'02 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, January 22, 2002 - Grenoble - IMAG

4.6 #I. H. Krüger, R. Mathew: Systematic Development and Exploration of Service-Oriented Software Architectures. Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), 2004.

4.7 #I. H. Krüger, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, S. Rittmann, J. Ahluwalia: Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering. Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS), 2004

4.8 Barbar Liskov, Jeannette Wing (1993). Family Values: A Behavioral Notion of Subtyping. Technical Report MIT/LCS/TR-562b.

4.9 Shaoying Liu (2004). Formal Engineering for Industrial Software Development. Springer. 3-540-20602-7

4.10 Tiziana Margaria (2004) Modelling Dependable Systems: What can Model Driven Development Contribute and What Likely Not? ISORC 2004, 7th IEEE Intern. Symposium on Object-oriented Realtime distributed Computing. May 12-14, 2004, Vienna (A), IEEE CS Press, pp. 113-120.

4.11 Tiziana Margaria, Bernhard Steffen (2003). Aggressive Model-Driven Development: Synthesising Systems from Models viewed as Constraints. The Monterey Workshop Series 2003 Theme: Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation, Chicago, Illinois, September 24-26, 2003.

4.12 J Mermet (editor) (2004). UML-B Specification for Proven Embedded Systems Design. The ChDL Series. Kluwer Acadmic Publishers. 1-4020-2866-0.

4.13 Nokia (2002). Mobile Internet Technical Architecture. Technologies and Standardization. ITPress. 951-826-668-9

4.14 Nokia (2002). Mobile Internet Technical Architecture. Solutions and Tools. 951-826-669-7

4.15 Nokia (2002). Mobile Internet Technical Architecture. Visions and IMplementations. 951-826-670-0

4.16 Object Management Group. Model Driven Architecture. http://www.omg.org/mda

4.17 Object Management Group. Unified Modelling Language. http://www.omg.org/uml

4.18 an Oliver (2004). Model Based Development and Embedded Systems Design. FDL04 Special Session on MDA in Embedded Systems Development.

4.19 Ian Oliver (2004) Some Issues in Rigorous System Design in a Model Driven Development Context. ECSI UML-SystemC System Design Flow Workshop. May 4, 2004, Lille, France

4.20 Ian Oliver (2004) Model Based Testing and Refinement in MDA Based Development. In: Proceedings of Forum on Design Languages FDL'04. September 14-17 2004, Lille, France.

4.21 Pankaj Palote (1998) Fault Tolerance in Distributed Systems. PTR Prentice Hall. 0-13-301367-7

4.22 J.Plosila, K. Sere and M. Waldén, Design with Asynchronously Communicating Components. In FMCO 2002: First International Symposium on Formal Methods for Components and Objects, Leiden, The Netherlands (November 2002), LNCS. Springer-Verlag.

4.23 M. Poppleton, R. Banach (1999) Retrenchment: Extending the Reach of Refinement. IEEE ASE-99, 158-165

4.24 Alastair D Reid (1993). A Precise Semantics for Ultraloose Specifications. MSc Thesis. University of Glasgow

4.25 Bran Selic, Garth Gullekson, Paul T. Ward (1994). Real-Time Object-Oriented Modeling. Wiley Professional Computing. 0-471-59917-4

4.26 Snook, C., Butler, M. and Oliver, I. (2003) Towards a UML profile for UML-B. Technical Report DSSE-TR-2003-3, Electronics and Computer Science, University of Southampton.

4.27 C. Snook, L. Tsiopoulos and M. Waldén, A case study in requirement analysis of control systems using UML and B. In Proceedings of RCS'03 - International workshop on Refinement of Critical Systems: Methods, Tools and Experience, Turku, Finland, June 2003. Also as: TUCS Technical Reports, No 533, Turku Centre for Computer Science, Turku, Finland, June 2003.

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

5.1. Introduction

The CDIS case study is aimed at providing feedback into the methodological and tool platform workpackages. While much of the initial work on this case study involves reviewing CDIS documentation, this process is extremely productive in providing challenges to the methodological research.

In early 2005 the first task of generating a subset of the original CDIS specification was completed. The first draft of the subset was discussed in a meeting at Newcastle on December 14th, 2005, where the degree to which things like concurrency and fault tolerance would be included was considered. The decision to draw the subset from a "vertical" slice of the original CDIS specification was generally agreed to be the best way to get a usable chunk of material.

The next meeting, held on January 20th and 21st, 2005, in Newcastle, was an opportunity to discuss the penultimate version of the CDIS subset. The entire context of CDIS was discussed, with respect to both the subset and the entire CDIS specification. A large part of the meeting was given to discussion of the problems initially faced during CDIS development, how they manifest in the subset, and the nature of the challenge the problems pose to the RODIN methodology. The January meeting also gave an opportunity to plan the first steps of how the subset would be redeveloped using the RODIN methodology.

The RODIN Plenary meeting at Nokia Research in Helsinki at the end of March gave a chance to present the subset and some of the work done on redevelopment to the wider RODIN community, and solicit feedback and view on the case study's progress.

5.2. Major Directions In Case Study Development

5.2.1. Nature of CDIS

CDIS was a technically successful project with exceptionally low defect rates for its time, and any new method must at least maintain the features that contributed to that success. At the same time, there were substantial technical difficulties which we would hope could be addressed by a more modern approach to the specification and design.

There are particular characteristics of CDIS that are relevant to the choice of method:

- CDIS is a data-intensive system. It contains large collections of disparate data and its behaviour is described by changes in that data. This is in contrast to, for example, control systems which are typically finite state and have only single instances of each kind of data.
- CDIS has a large external interface. Indeed its whole purpose is to display information. The specification is therefore large in terms of the number of operations and the number of inputs and outputs for each operation.
- CDIS does little processing. There are almost no interesting algorithms in CDIS, and the specification is shallow: each operation has a simple though voluminous definition.
- CDIS is a highly distributed system. It exhibits a high degree of concurrency, and the idea of an atomic operation is a very loose approximation to its actual behaviour.

There are five major areas that the case study needs to address:

1. Comprehensibility
   Any system specification needs to be understood by all the stakeholders. The formal notations[1] in CDIS were a barrier to understanding by many stakeholders.
2. Modularity
   The original CDIS specification and design take thousands of pages. A good modularity mechanism is essential.
3. Concurrency
   (a) We have no formal ways of reconciling the atomic operation model with the real concurrent behaviour
   (b) We needed different, unrelated notations for sequential and concurrent specifications.
4. Refinement
   Conventional models of refinement are quite inadequate to express the change of structure between the specification model and the design model.
5. Proof
   It was completely infeasible to carry out proofs on a specification of this size.

5.2.2. Comprehensibility

The case study needs to ensure that the new specification is comprehensible to stakeholders. It needs to be capable of expressing stakeholder concepts in a direct way rather than having to translate them into some obscure (even if faithful) representation. For example, notions such as aggregation, sequencing and optionality of data must be directly expressible in the specification language.

[1] Principally VVSL — VDM augmented with modularization constructs

$DELETE\_PAGE(pageno : Page\_number)$
$edd\_dspl\_updates : EDD\_displays$

ext wr $pages : Pages$
wr $version\_sets : Version\_sets$
wr $checked\_out : Checked\_out$
wr $edd\_pages : EDD\_pages$
wr $edd\_displays : EDD\_displays$
rd $page\_selections : Page\_selections$
rd $preview\_selections : Preview\_selections$
rd $time\_now : Date\_time$

pre true

check $can\_delete\_page(pageno, \overleftarrow{pages}, \overleftarrow{version\_sets})$

post $page\_deleted(pageno, \overleftarrow{pages}, pages,$
$\overleftarrow{version\_sets}, version\_sets)$
$\wedge$
$page\_deleted(pageno, \overleftarrow{edd\_acks\_required}, edd\_acks\_required,$
$\overleftarrow{concealed\_displays}, concealed\_displays,$
$\overleftarrow{edd\_displays}, edd\_displays, \overleftarrow{edd\_pages}, edd\_pages,$
$edd\_dspl\_updates, page\_selections, preview\_selections,$
$pages, time\_now)$

Figure 5.1: The VVSL specification for Deleting Pages

5.2.3. Modularity and size

VVSL [5.6] has a modularity mechanism similar to VDM-SL [5.2]. It allows types, state and functions visible in one module to be used in another. It also contains a good mechanism — perhaps better than any other specification language — for expressing error behaviour. In spite of this, the CDIS specification is extremely verbose. The main reason is that there is no good mechanism for modularising the definition of operations. Figure 5.1 is an example of an operation definition from the subset. This specification is simply importing the three partial definitions, but it requires a huge amount of boilerplate to make it a correct VVSL specification. There is currently no completely satisfactory solution to this problem. For example, while

DeletePage
DeletePageFromPages
DeletePageFromDisplays

CanDeletePageFromPages

Figure 5.2: Z Schema for Deleting Pages

the schema calculus in Z [5.7] has limitations, although the corresponding Z specification in Figure 5.2 would certainly be more palatable.

Any new notation needs to allow specifications that are closer to Figure 5.2 than Figure 5.1.

5.2.4. Concurrency

It is obviously useful to separate concerns about behaviour from concerns about concurrency. However, it is also necessary to bring together the two views of the system, and in CDIS we weren't able to do that very effectively. There are two issues: the approximate nature of the VVSL specification and the incompatibility of notations.

5.2.4.1. VVSL versus reality

The VVSL specification of the operation to release page states that all workstations that are displaying the page simultaneously switch from displaying the old version to displaying the new one. This is not necessarily what the system actually does: the different workstations may take varying lengths of time to do the switch. There are many similar examples in CDIS. The methodological challenge that this poses is whether there is a justifiable retrenchment from the strict specification to some looser (and presumably more complex) specification that describes the real behaviour. The reason this is so desirable is that the initial strict specification is relatively simple and captures the essence of the operation; the complexity of non-simultaneous update is really a separate issue. Furthermore one would like to have a single generic way of retrenching all the different operation specifications, rather than cluttering each one with the details of non-simultaneity.

5.2.4.2. Notations

Although VVSL does have a notation for concurrency, it was not in practice usable. Therefore we had to use quite separate notations such as CSP or CCS (as well as informal notations like data flow diagrams) to describe concurrency. There is therefore no formal connection between the sequential and concurrent specifications. The case study needs to demonstrate that Event-B can integrate the different aspects while still allowing different concerns to be treated fairly independently.

5.2.5. Refinement

VDM [5.5] (on which VVSL is based) and Z have a simple model of refinement in which state can be made more concrete and operations can have their preconditions weakened and post-conditions strengthened. This notion doesn't begin to address the complexities of design of a

distributed concurrent system like CDIS. Nor does it capture the idea of viewing an operation at different levels of granularity, which is essential if the initial specification is not to be cluttered with too much detail.

5.2.5.1. Concurrency and distribution

State is split over many machines; operations involve several different processes and messages between processes and machines, so the set of operations in the design is far richer than the set of operations in the specification. A single specification level operation may require multiple instances of several design level operations and messages, while a single design operation may contribute to many different specification level operations. We need a calculus of refinement that allows such structural changes to be expressed.

5.2.5.2. Granularity

At one level, we do not worry about how inputs to operations are provided. The VVSL operation to display a page, for example, has an input of type *Page_Number*. At another level, however, we need to specify exactly how this input is derived from user actions. For example, the page number is typed on a special keypad and there is quite a complex protocol to allow the user to change their mind or do something else in the middle of selecting a page. We did not have any formal way of relating these two views of the operation. The protocol was described as a finite state machine but its connection to the VVSL was entirely informal.

5.2.6. Proof

As CDIS is a shallow but large system, there is a huge volume of relatively trivial proof obligations. It is important that the tools are capable of carrying out the necessary proofs with a minimum of human intervention, since the sheer volume would make it impractical to do the proofs by hand.

Of course, while the specification level proofs are trivial, proof obligations introduced by the more complex refinement rules mentioned in section 4 may not be. There will also be a very large number of such proof obligations.

5.3. Achieved Results

There are two major results to date in this case study. The first is the subsetting of the original CDIS specification to a usable set of documents for the project. The rationale behind the selection is described in the next section.

The second major result is our work on the redevelopment of CDIS in the RODIN notations. This work has been done at Praxis and Southampton, each following a different approach. By the plenary meeting at Helsinki we decided that the approach initially taken at Praxis would not be the best option. The reasons for choosing not to redevelop the subset first in classic B [5.1] notations are given in Section 5.3.2.

5.3.1. Content and rationale for the CDIS subset

CDIS is an operational system supplying flight data, airport data and other support information to air traffic controllers in the London Terminal Control Centre at West Drayton. It was delivered in 1992 and went operational in 1993. Since then it has been maintained by Praxis and subsequently by NATS: some parts of the original system have not been used, others have been changed and new facilities have been added. The subset used in this case study is based on the system that was originally delivered.

There is a brief overview of CDIS and a description of the specification and design approach used in an IEEE Software article [5.3]. The subset is fully described in a project document [5.4].

The RODIN subset includes only airport data. The subset has been chosen to be small enough to be manageable while retaining some of the main features of the CDIS specification and design. In particular it includes enough to illustrate:

- the distributed nature of CDIS
- the necessary size of the specification, which required us to spread the definition of operations over several modules;
- the existence of concurrency in the inputs to CDIS, and the extra concurrency introduced by the distributed design; and
- the need for several different aspects (user interface, functional and concurrency) of specification and design.

The subset documents are:

- Requirements
  - Semi-formal requirements definition including tracing
  - Interface control document
- Specification
  - A core specification written in VVSL [5.6], where each operation is modelled as an atomic change of state.

  - A concurrency specification that describes where the atomicity assumptions of the core specifcation breaks down.
  - A user interface definition that describes the concrete appearance of inputs and outputs.
- Design
  - Application design which defines how the functionality of the core specification is implemented.
  - Process design which describes the behaviour of each concurrent process in the implementation.
  - Specification of low-level services relied on by the implementation.

5.3.2. Redevelopment Work

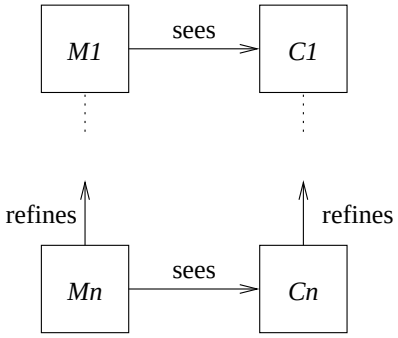
Southampton has been looking specifically at the requirements document and specification documents of the Praxis case study. Therefore, at this stage, we have ignored all design and interface documentation. The aims of this case study are somewhat different from the other case studies because our goal is to use the development of (a subset of) an existing air-traffic control system to assess the performance of the methodologies evolving within the RODIN project. Hence, there is an immediate conflict: in order to assess the new methodologies, we need to be able to compare our models with those of the original development, but the methodologies (in particular Event-B) do not necessarily facilitate this assessment.

Even though classic B has many similarities with VDM (the underlying specification language for CDIS), and there has been work done on converting one notation into the other, Event-B is a fairly radical departure from its ancestor. This is especially true for the structuring of large specifications. At Southampton, therefore, we do not consider a 'translation' to play a significant part in this case study. Instead, we propose to use the original specification as a guide to refinement from a more abstract viewpoint. This will also allow us to highlight differences between Event-B and classic B.

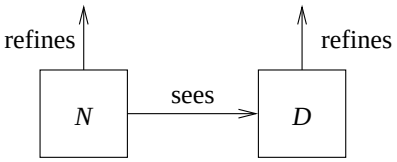
The main features of the CDIS specification is its size and complexity. While it is inevitable that large real world problems will produce large and complex specifications, facilities should be provided by the specification language for managing such complexity. These structuring mechanisms have a significant influence on the development process. In Event-B, two approaches to the structuring of large specifications have been proposed: generic instantiation and decomposition. In the next two sections, we consider whether these approaches can be applied to CDIS.

5.3.3. Generic Instantiation

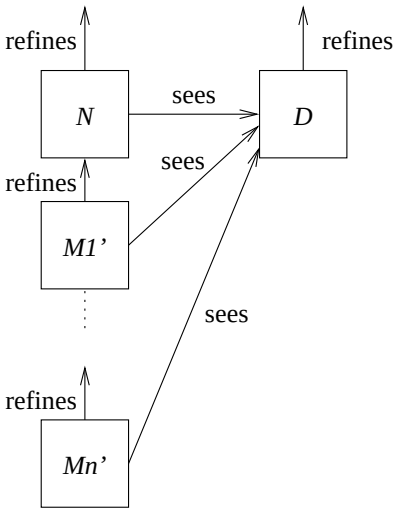
Refinement is the means by which more and more complexity is added to an abstract specification in order to move formally towards an implementation. Rather than performing a single



Development 1



Development 2



Instantiation

Figure 5.3: An instantiated refinement of development 2

linear refinement, it is possible to perform separate refinements which can be combined at a later stage.

A formal development in Event-B is said to be *generic* with respect to the sets and constants accumulated during the development because these can be viewed as parameters to the development. By instantiating these sets and constants with the sets and constants of a second development, we can get an *instantiated refinement* of the second development via the refinement steps of the first development providing certain proof obligations are fulfilled.

Informally, the proof obligations require:

- the properties of the constants of the second development to imply the (instantiated) properties of the first development,

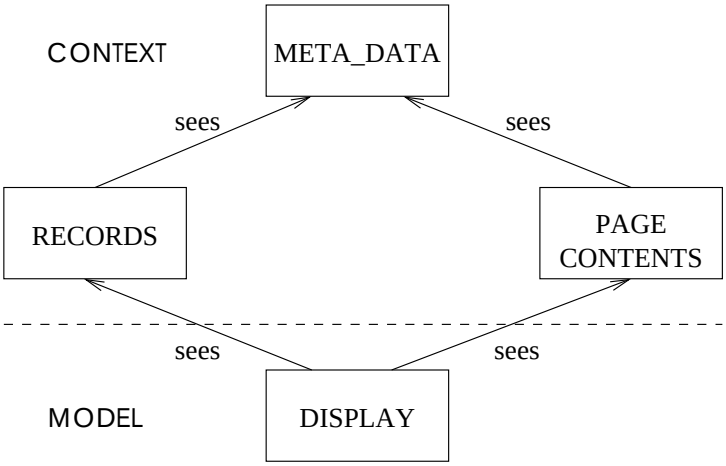


Figure 5.4: A model for a generic display

- the most abstract model of the first development to refine the most concrete model of the second development.

This is depicted in Figure 5.3 (where the models M_1', ..., M_n' are the instantiated models M_1, ..., M_n).

By using the core specification of CDIS as a guide, we can start from a more abstract viewpoint by constructing a model of a generic display system. The context for this generic model is made up of a hierarchy of three sub-contexts and one model (see Figure 5.4). The three sub-contexts are generalisations of the modules AIRPORT META_DATA, AIRPORT RECORDS and AIRPORT_PAGE CONTENTS in the core specification. By abstracting away the airport-specific details, we get a clearer picture of the display functions of the system. At a later stage in the development, we can instantiate this model with the airport-specific details from which the Event-B model can be refined.

- **META_DATA**. In the core specification, this module defines types for attribute identifiers and values for the airport and runway data. Here, we declare generic types for attribute identifiers and values in the **SETS** clause
  - *Attr_id*
  - *Attr_value*
- **RECORDS**. By including META_DATA via a **SEES** clause, we define a generic type for records as a mapping from *Attr_id* to *Attr_value*
  - $Records \in Attr\_id \rightarrow Attr\_value$
- **PAGE CONTENTS**. Here, we define types for the layout of generic pages. We consider each page to be made up of a background and a number of fields corresponding to the

position and appearance of values of attributes[2]

- *Page_contents* :: *background* : *Graphic_background*
  fields : $\wp(Graphic\_field)$

- *Graphic_field* :: *id* : *Attr_id*
  posn : *Field_position*
  presn : *Attr_value* $\rightarrow$ *Fld_disp_desc*
  The type *Fld_disp_desc* is meant to represent the device-independent 'appearance' of fields. Hence, the function *presn* maps values (for a particular attribute) to their appearance.
- *Page_disp_desc* is a type representing the device-independent 'appearance' of a page. The intention is to combine a background with a set of *Fld_disp_desc* (together with their respective positions) to form a value of type *Page_disp_desc* that represents the value of a single page.

Given the state that defines actual records, page contents and displays, it is then possible to define events that compute the appearance of displays. This is depicted as the model DISPLAY in Figure 5.4.

Once this development has been shown to behave as expected, it would be possible to instantiate this with specific airport attributes and values from which further refinements can be made.

5.3.4. Decomposition

This approach aims to reduce the size of a model by *decomposing* it into a number of smaller sub-models. This involves partitioning the events and variables so that the events of one sub-model do not affect the variables of the other sub-models. Hence, decomposition aims to *distribute* the model so that certain elements can be seen to be 'remote' from other elements.

Communication between distributed elements is achieved through the use of *shared* (or *external*) variables (i.e. state variables that are common to more than one element). Unfortunately, the CDIS core specification does not exhibit this structure. It is more reminiscent of the **INCLUDES** mechanism in classic B, in which an operation of the including machine can call separate operations of the included machines to update their respective states. However, if we are prepared to deviate from the structure of the original specification then it is possible to make use of Event-B's decomposition mechanism.

It is possible to decompose the DISPLAY model given in Figure 5.4 into three sub-models: a records model, a page contents model and a (new) display model (this is shown in Figure 5.5). One can imagine partitioning the events in DISPLAY accordingly: record-updating events belong to the records model, page-updating events belong to the page contents model, and display

[2]The record notation used here is part of our proposal to extend B with record types. Details of this can be found in the D9: WP2 Deliverable D2.1 document.

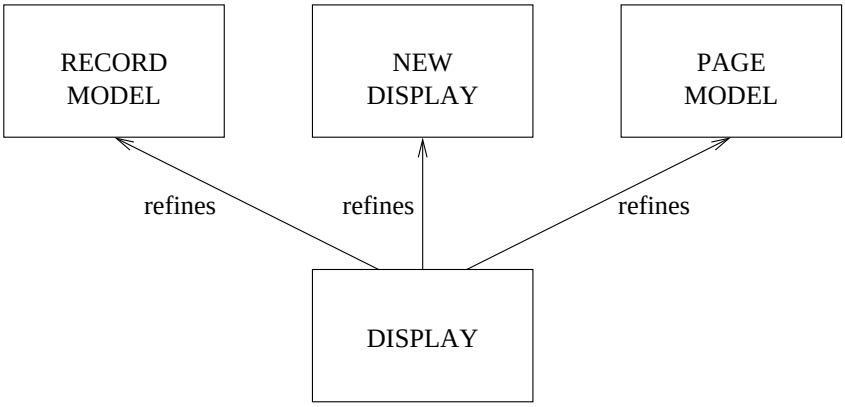


Figure 5.5: A decomposition of DISPLAY

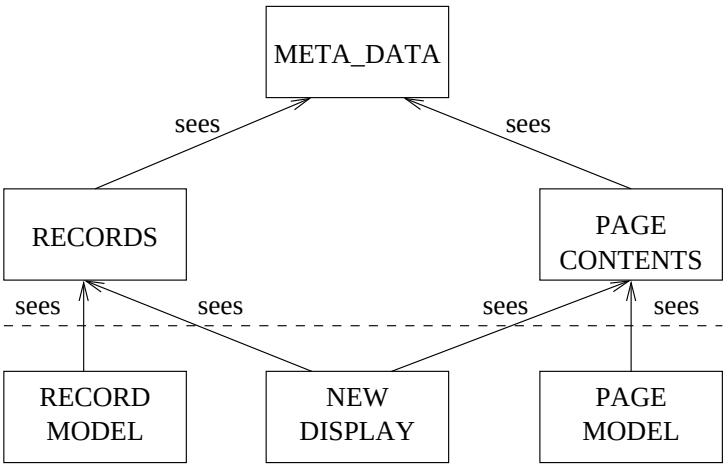


Figure 5.6: Putting it together

events belong to the display model. In this decomposition, the shared variable is the state representing the actual displays because any attribute update will affect the display of its value, and similarly, any update of the page contents will also affect the display. The combination of the context of Figure 5.4 and the decomposed model of Figure 5.5 is shown in figure 5.6. Note that the new models only need to see a subset of the context. That is, RECORD MODEL does not need to see the PAGE CONTENTS sub-context, and PAGE MODEL does not need to see the RECORDS sub-context. NEW DISPLAY requires both sub-contexts.

5.3.5. Other Issues

Another source of complexity within the CDIS core specification is the abundance of union types. Whenever these types are used as parameters of functions, the body of the function inevitably comprises a complicated case statement which is difficult to understand. During the early stages of the development it would be beneficial to keep these component types separate, and to define specialised events for each case. Indeed, the bodies of events in Event-B do not allow branching.

The VVSL extensions that enable error handling within functions as well as operations are not present in B. The specialisation of events appears to be one way in which error handling can be incorporated into an Event-B specification. One can then, for example, define events corresponding to exceptional circumstances in order to meet the robustness requirements of the system.

5.4. Demonstrators

The Event-B models and specification document for the CDIS subset form the tangible demonstrator for this particular case study. These models, when used with the RODIN tools, should demonstrate the tools' applicability and usefulness with respect to industrial-sized projects.

5.5. Future Development

The CDIS case study has already proved to be very beneficial to the project. Although it contains only a relatively small subset of the original CDIS specification, it is nonetheless larger than most research case studies and it is rich in "real world" complexities.

Less progress has been made in reworking the CDIS specification and design using the RODIN-nominated notations (Task 1.4.4) than we might have been wished; however, (as in all scientific experiments) it has been the difficulties and failures that have been the most instructive. In particular, it has become clear that any naive attempt at direct translation of the CDIS VVSL specification into Event-B will simply reproduce the size and complexity deficiencies of the original in a new form.

Future work on Task 1.4.4 will not pursue that approach further but will, instead, focus on further development of the work started at the University of Southampton as described in section 5.3.2. This approach attempts to make use of the possible new strengths of the Event-B notation to create a new specification from the CDIS requirements. The existing CDIS specification will serve a guide for the refinements needed rather than as a detailed route map. The strength of this approach is that it will much better serve to evaluate the benefits of Event-B. A potential drawback is that the resulting specification might be quite different in form from the original VVSL specification, thus making it harder to show that the Event-B version is in some way equivalent to it. The continuing work in this area will provide inputs to further methodological investigations (WP2) and the tool development tasks (WP3 and WP4).

5.6. References

[5.1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[5.2] J. Dawes. *The VDM-SL Reference Guide*. Pitman Publishing, 1991.

[5.3] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, March 1996.

[5.4] A. Hall. Description of CDIS subset. RODIN/Praxis internal document S.P1286.50.2 Issue 0.3, January 2005.

[5.5] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall, Inc., 2nd edition edition, 1990.

[5.6] C. A. Middelburg. VVSL: a language for structured VDM specifications. *Formal Aspects of Computing*, 1:115–135, 1989.

[5.7] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

SECTION 6. INITIAL REPORT ON CASE STUDY DEVELOPMENTS FOR CASE STUDY 5: AMBIENT CAMPUS – THE LECTURE SCENARIO

6.1. Introduction

The overall project work on the Ambient Campus case study is focusing on

- elucidation of the specific fault tolerance and modelling techniques appropriate for *Ambient Intelligence* (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.

The first year our work has focused on three major subtasks (see Project Description of Work [6.1]):

T1.5.1. Define case study, evaluation plan, measurements and assessment criteria
T1.5.2. Produce informal specification of the ambient campus; identify problems related to provision of fault tolerance
T1.5.3. Identify general solutions from the area of application level fault tolerance (such as atomic actions and exception handling) to be adapted to AmI applications; apply adapted techniques to the ambient campus system.

6.2. Major directions in case study development

During the first year we have been mainly working on the first scenario – the Ambient Lecture scenario. This chapter provides a progress report on the Ambient Campus case study, in particular regarding the lecture scenario. After the completion of the *requirements document* for the Ambient Campus case study [6.2], two strands of work have been carried out. One involves the development of modelling techniques that support rigorous design of the lecture scenario using the *B method* (see sections 6.3.2 – 6.3.3). The other strand focuses on the feasibility study of the hardware and software that can be used for implementing this system (see sections 6.3.4 – 6.3.5).

In the second year, we will work on rigorous specification of this scenario, on its formal modelling using one of the RODIN formalisms (most likely B), on application of the mobility

abstractions which are under development in WP2 and on preparation of the second scenario. We are planning to apply the general refinement/decomposition techniques to be developed in WP2 and to progress further with our programming experiments. It is our plan to develop a prototype demonstration to show it during the 4th RODIN workshop in April 2006.

Later on, we will develop the second scenario, evaluate the applicability of the process-based modelling techniques (WP2) in this case study and apply the mobility plug-in (WP4) to model-check mobility and fault-tolerance specific properties of the Ambient Campus scenarios.

The rest of this chapter discusses in detail the progress that has been made, along with the recommendations for further work to be done.

6.3. Achieved results

The first major result is the development of the requirements document written to capture the main characteristics of the system for supporting the lecture scenario of the Ambient Campus case study [6.2]. This system is meant to support mobile devices such as Personal Digital Assistants (PDAs) communicating with each other through wireless network supported by hotspots. The use of wireless network provides a challenge – among others – in the way communications are conducted, where loss of communication might become a common issue. This leads to a requirement for the system to tolerate and handle communication losses appropriately. An overview of the requirements document is given in section 6.3.1.

In order to implement the Ambient Campus system, a formal approach for designing the architecture for supporting mobile devices is taken. This design is performed using the B method and through several refinements, the *Context-Aware Mobile Agents* (CAMA) system specifying the scenario is formulated. More details on this system and its development process are discussed in section 6.3.2.

Mobile device potentials are not fully exploited without introducing *mobile agents*. In our case, a mobile agent is a representation of the human user; it is a piece of software that allows a human user to interact with the services provided on a wireless location through his/her mobile device. Our work on developing the techniques for formal specification of mobile agents and their functionality is summarized in section 6.3.3.

Considerable amount of work has also been done on the preliminary exploration of the mobile devices and the underlying technologies that are to be used in the lecture scenario. These include the hardware (such as PDAs and hotspots) as well as the software (such as the middleware for supporting mobile agents, programming language and platform support) that are needed for the implementation and the demonstration of the lecture scenario. Sections 6.3.4 and 6.3.5 cover the hardware and software technologies respectively.

6.3.1. Requirements document

Based on the methodological approach outlined by J.-R. Abrial [6.3], we prepared a

requirements document for the Ambient Campus case study. This document focuses on the lecture scenario. In particular, we introduce *Ambient Campus Environment* (ACE) concept to cover entities that are needed to support the running of the lecture scenario. These include the PDAs, desktop computers, software agents (to be run on the PDAs and desktop computers), as well as hotspots to enable wireless communication among the software agents.

For better understanding and clarity, the requirements are classified into several categories. The taxonomy of the lecture scenario requirements is as follows:

1. **EN** – this requirement covers general requirements regarding the *environment*, including statements on the required properties of users and ACE.

2. **FT** – this requirement deals with *fault tolerance* issues: the system should be able to tolerate a number of abnormal situations such as connectivity loss, failures of PDAs and desktop computers, violation of time constraints and fire alarms. These requirements define a set of related abnormal situations.

3. **ST** – the requirement of this category specifies the *states* that an agent can fall into, and how the agents change their states depending on the role or activity they are performing at a particular time.

4. **SV** – this requirement defines *service requirements and restrictions* that ACE should provide.

5. **QL** – the QL requirement sets additional requirements related to the *quality of service*, such as performance and resource usage that certain services provided by the agents need to satisfy.

6. **SE** – this requirement captures issues related to *security*, such as access permission, authorization and shared resource access.

7. **TT** – the requirement of this category lists the *delays and timeouts* associated with various services or service quality requirements.

Using this taxonomy, the full requirements were constructed; readers are recommended to consult [6.2] for a thorough description and full explanation of these requirements. In this section, we show some important excerpts from the requirements document.

6.3.1.1. Location and connectivity

Location refers to a room on campus that has a hotspot providing wireless connectivity. Connectivity area may reach beyond the room or even cover several rooms. Connectivity areas may also overlap (Figure 6.1).

The users (i.e. people) involved in the scenario are teachers and students. There is one teacher and several students involved in each lecture activity. Teacher and student users are represented by software agents in the ACE: the teacher agent is run on the desktop computer present in the

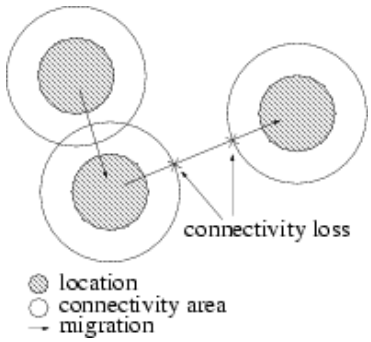


Figure 6.1: Location and connectivity area

room whereas each student agent is run on an individual PDA (see Figure 6.2). The users control the actions of their agents through a graphical user interface presented on their device (desktop computer or PDAs).

When a student carrying a PDA moves from one location to another, there is a possibility that his/her agent will experience a loss of connection, and re-connection later (see Figure 6.1). This should be handled appropriately by the system.

6.3.1.2. Possible lecture activities

One of the most important aspects of the scenario is the representation of the activities that can be performed during a lecture. Typical activities might include:

- Registration of students
  At the beginning of a lecture, the teacher takes attendance of the students. Students register to the lecture through their PDA, and there is an authentication process to ensure that only authorised students will be allowed to participate in the ACE-supported lecture scenario.
- Material dissemination
  Teacher can distribute lecture related materials, such as lecture notes, reading list or timetable/venue changes directly to students' PDAs.
- Organisation of students into groups
  Students may be organised into groups to do a group task or to allow group discussions. Enforcement of group policies (such as only those within the same group can talk to each other) will be observed in the scenario.
- Individual task
  From time to time, teacher may give students a task to do individually. This includes those tasks for assessing the progress of each student, so we need to ensure that the submissions from the students are handled appropriately.
- Group task
  The teacher may also give tasks to be performed through collaborative effort by students in groups. As with the individual tasks, the submission of the group tasks must be handled appropriately.

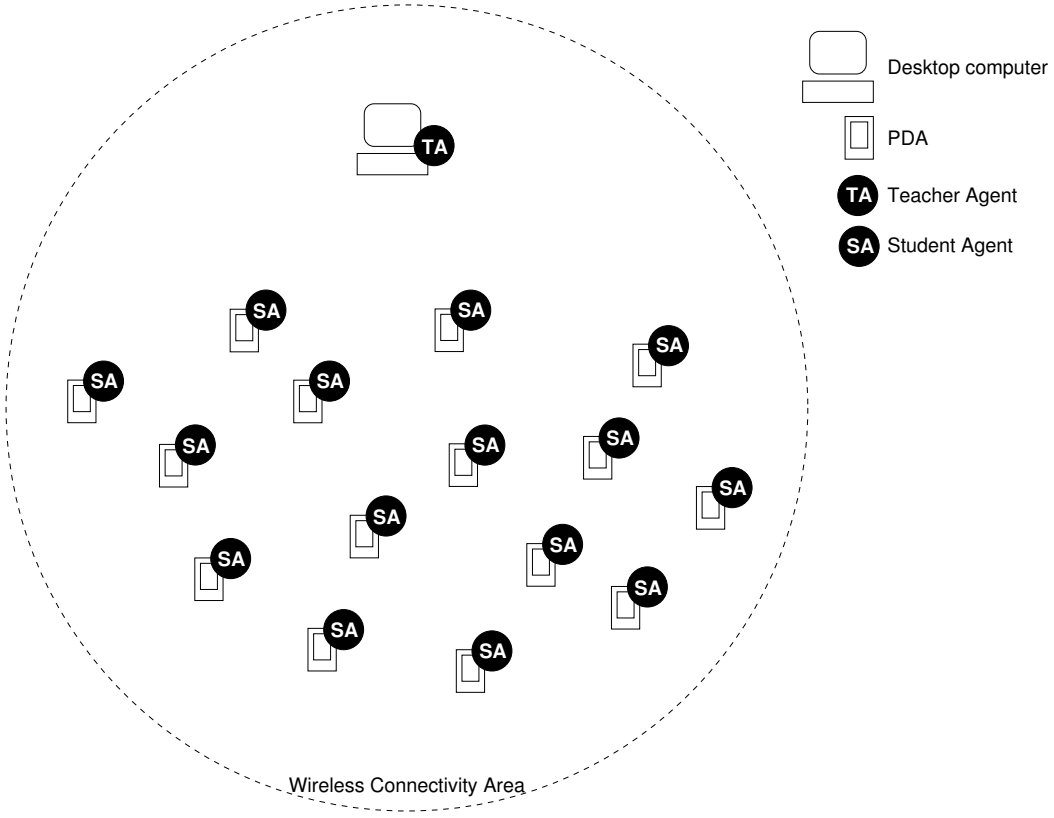


Figure 6.2: Arrangement of Ambient Campus Environment

- Questions from students
  Students might be more inclined to ask questions on materials they do not understand if they can do it quietly through their PDA. The teacher can then answer the questions privately or if he/she thinks it will benefit the whole class, the answer can be broadcast to all. The teacher could also pass the questions to the class to see if any of the students can provide an answer.

More activities can possibly be added later, but for now we think the above set should provide enough challenge for developing and testing the software system to support the lecture scenario.

The requirements document also addresses emergency, failure and timing requirements, these can be found in [6.2].

The following sections discuss the framework and infrastructure for implementing the Ambient Campus case study.

6.3.2. CAMA system

The *Context-Aware Mobile Agents* (CAMA) system consists of a set of locations and basic coordination functionality provided by the middleware. Active entities of the system are agents.

An agent is a piece of software that conforms to some formal specification. The requirements for agents are developed using the B Method. Each agent is executed on its own platform. The platform provides an execution environment and an interface to the location middleware. Agents coordinate their execution through a Linda-type coordination *tuple space* to ensure their asynchrony and anonymity. More specifically, they communicate through the special construct of the coordination space called *scope*, which structures their coordination. An agent can cooperate only with other agents through participation in a common scope. Agents can logically and physically migrate from a location to another location. Migration from a platform to a platform is also possible using logical mobility. An agent is built as a combination of one or more roles. Each role is a formal specification of a specific functionality so that a composition of specifications of all the roles forms the specification of the agent. Each role is a result of the decomposition of an abstract scope model so that each concrete run-time scope is an instantiation of an abstract scope model.

6.3.2.1. Formal development of CAMA system

The formal development process of the CAMA system consists of several steps. First, we create abstract specifications of the middleware (location) and the scopes to be supported by the system. Then we develop (by the stepwise refinement method) specifications of different roles participating in scopes. Finally, we compose an agent specification as a combination of several developed roles (i.e. agent interfaces) and the default functionality defining the agent behaviour outside scopes.

Agent specification can be further refined by adding more details and custom functionality. Communication compatibility of different agents is ensured by the fact that all agents are developed by the formal refinement method from the same abstract specifications of different roles and the middleware. Therefore, agents can collaborate making safe assumptions about the functionality of their peers.

The B Method [6.3] (further referred to as B) is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [6.4], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called refinements. A formal specification is a mathematical model of the required behaviour of a (part of a) system. In B, a specification is represented by a set of modules, called *abstract machines*. An abstract machine encapsulates state and operations of the specification and as a concept is similar to a module or a package.

We have started some initial work on application of the model-checking mechanisms to verification of agent properties. In particular, we are interested in the behaviour of communicating multi-agent systems, including their fault tolerant aspects. For this purpose, we are developing the CAMA semantics and multi-stage translation process from CAMA to Petri nets. This work is carried out as a joint activity between WP4 work on mobility plug-in and WP1. In our approach [6.5], *Klaim* process algebra [6.6] and pi-calculus will be employed for the intermediate representation and transformation into Petri nets. The model checking technique adopted in our work is a partial-order model-checking based on Petri net unfoldings.

6.3.3. Mobile agent development

In our approach, an agent specification is built by extending one or more roles obtained formally through decomposition of the abstract scope models. The refinement step introduces a specification of the minimal agent functionality called the default role. This functionally permits an agent to talk to locations, create/join/leave scopes, and do migration. Agent may also need some logic that glues independent interfaces and allows them to talk to each other. This is done via global agent variables and the special methods for accessing them.

Upon its completion, the agent specification is used to build the source code for the actual agent program. The source is linked with the middleware library to get an executable agent program. The generated agent source may run on PDAs, laptops, desktop PCs and smart-phones using the platform-specific middleware implementation as the adaptation layer.

The standard work cycle of an agent looks like this: an agent detects the available locations and connects to at least one of them, then it looks for the current activities on the location(s) or creates its own new scope, and finally it joins a scope and plays one of the implemented roles in it. Only when the agent decides to play a particular role in a scope, it really starts to cooperate with other agents. The agents are interoperable and able to understand each other because the role functionalities of all scope participants are based on the same abstract model. As a result, the composition of agent functionalities in a scope corresponds to the initial abstract model.

6.3.4. Mobile technology: hardware

Before we can develop any software agents, we need to familiarise ourselves with mobile technologies, including the hardware. This led to the purchase of several PDAs and a pair of wireless hotspots.

The PDAs must have features for supporting wireless connection and they must be able to run Java programs. In the end we decided to purchase four HP-IPAQ hx2750 running Microsoft Windows Pocket PC 2003. This model is near the top of the range line at the time.

We could in theory just “piggy back” on the wireless network provided by the School of Computing Science at the University of Newcastle through their hotspots, but we would like to experiment with different configurations of wireless hotspots. One configuration is where the hotspot is connected to just the desktop computer present at a location; another is where the hotspot is connected to the campus wide network.

We also obtained a pretty basic desktop computer on which we installed Linux operating system. This computer hosts the middleware (i.e. the location or tuple server) needed to coordinate the communication among software agents. A similar installation in the lecture scenario will also be used by teachers to deliver lecture, i.e. it will serve as a platform for hosting the teacher agent.

6.3.5. Mobile technology: software

We need customised software to provide the infrastructure for ACE such as the tuple space server (middleware), as well as the agent software to run on the PDAs.

6.3.5.1. Architectural comparison

For the middleware, we are using the asymmetric scheme which is closer to the traditional service provision architectures. This scheme is based on the concept of a fairly reliable infrastructure-provided wireless connectivity. The alternative symmetric scheme can also operate in ad-hoc networks and all the coordination functionality is implemented by the agents.

In our scheme the larger part of the coordination and controlling logic is moved to a location server. This approach can support large-scale mobile agent networks in a very predictable and reliable manner. It makes the better use of the available resources since most of the operations are executed locally. Moreover, the asymmetric architecture eliminates the need for complex distributed algorithms or any kind of transactions. This allows us to guarantee atomicity of certain operations without sacrificing performance and usability. The scheme also provides a natural way of introducing context-aware computing where a location is part of an agent context. The major drawback of the asymmetric scheme is that an infrastructure support is always required for mobile agents to collaborate.

6.3.5.2. Middleware: location/tuple server

After the initial experiments with several mobile agents middleware systems [6.6][6.7][6.8], we decided to start the development of our own mobile agents middleware. The major rationale for this decision is that all these systems implement ad-hoc style communication which we found to be too heavy-weight for the available hardware. Another problem is the lack of industrial strength implementations of ad-hoc routing algorithms. Our approach is based on the location-based architecture where all the coordination in a location happens inside a single coordination server. This allows us to achieve better performance and enrich the concept of coordination space with structuring, exception handling and security features.

6.3.5.3. Java platform for PDAs

The software agents are to be written in Java. In order to run Java applications on the PDAs, we need to first install Java Virtual Machine (JVM) on them. We investigated several JVMs that are available for Pocket PC 2003. There are not many such JVM available, we tried (the demo version in the case of those commercial ones) and compared several of them:

- eWe from Ewesoft [6.9]
  eWe is a reasonably well-developed and freely available JVM, but it does not have standard Java libraries, therefore there are some important features missing (such as "serialization").
- J9 from IBM [6.10]
  This JVM supports standard Java libraries bar several (for example, swing) and available to purchase for a reasonable price.
- CrEme from NSIcom [6.11]
  This is one of the most comprehensive JVMs for PDA, but on the down side, it is very expensive to purchase.

We decided to use IBM's J9 to provide Java platform on our PDAs. J9 is chosen because it supports all the necessary features that we intend to use for our software agent development and it is not too expensive to purchase ($5.99 per license).

6.3.5.4. Example

We have developed a simple system to demonstrate the feasibility of ambient applications using tuple space and PDAs where the communication is done through a wireless network. This system allows students and a teacher to perform text-based chatting; in this application, the students use their PDA, and the teacher uses a desktop computer, which also hosts the tuple space server.

The Java implementation can be divided into two main parts:

- *jcama* library providing classes for dealing with tuple space through CAMA architecture (see section 6.3.2). At the moment, only basic `in` and `out` operations are supported, allowing objects (such as strings, numbers and even binaries) to be placed into and taken out of the tuple space. The final version will provide all the communication primitives along with the scoping mechanism, exception handling and migration. We are also planning to port the client-side middleware to other platforms and languages, such as Symbian smartphones, .NET and Python.
- Graphical User Interface (GUI)-based agents. There are two types of agents: student and teacher agents.
  - The student agent allows student to type in a message and to send it to the tuple space. This agent also checks the tuple space for any message addressed to it and displays this message on the PDA (as chat history) when that happens.
  - The teacher agent manages the tuple space. It picks up any message sent to the tuple space and broadcasts it to all agents present at that time. Teacher agent can send messages as well. It also keeps a record of all student agents currently present, along with the chat history.

  In the client-server terminology, the student agent can be considered to be the client whereas the teacher agent, the server.

There can only be one teacher agent running at a given time, but there can be multiple student agents. Each student agent has a unique identifier, which enables the teacher agent to distinguish

them. When a student agent is run, it tries to detect if there is a teacher agent present. If that is the case, the student agent will join the chat session. If there is no teacher agent running, the student agent cannot send any message to the tuple space. When a teacher agent is activated, it detects any student agents currently running and automatically makes them join the chat session. Agents can join or leave a session at any time; the system provides the functionality to handle these events accordingly.

Screenshots of this system are given below. To obtain these screenshots, all of the agents were run on a desktop computer, but in practice, student agents are usually run on PDAs. Since these agents are written in Java, they can be easily ported on various platforms or devices.

Teacher runs his/her agent, and when students run theirs, they will be automatically added into the session and they will receive a welcome message. Teacher agent lets the teacher know who just joined the session (see Figure 6.3). Student agents can join at anytime, as long as the teacher agent is running.

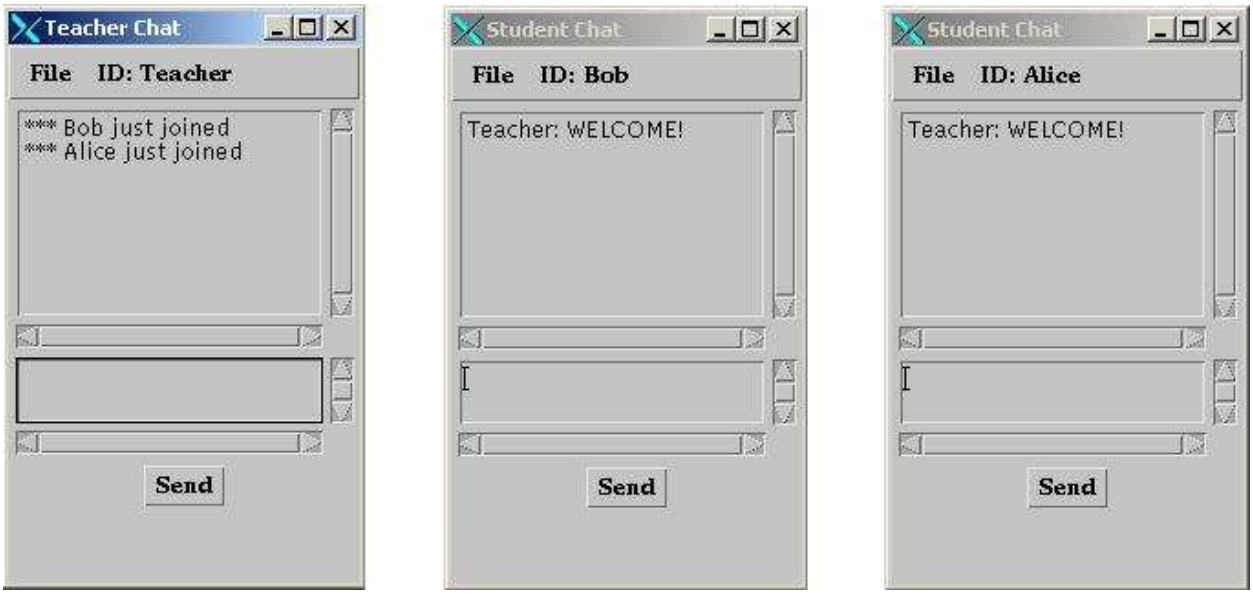


Figure 6.3: Initiation of the chat session

Teacher and students can then exchange messages through their agents, as can be seen in Figure 6.4. This screen capture also shows that one student ("Bob") is about to leave the chat session by invoking the "Quit" action. When this happens, teacher agent will display a message to let the teacher know that a student had just left the chat session, but other students will not know, and the session just continues (Figure 6.5). It is pretty straightforward to modify the system so that all agents will be notified if anyone leaves. When the teacher agent leaves the session, the chat session is terminated. There is nothing else that the student agents can do apart from leaving (quitting) the session, unless the teacher agent comes back and rejoins, upon which the chat can be resumed.

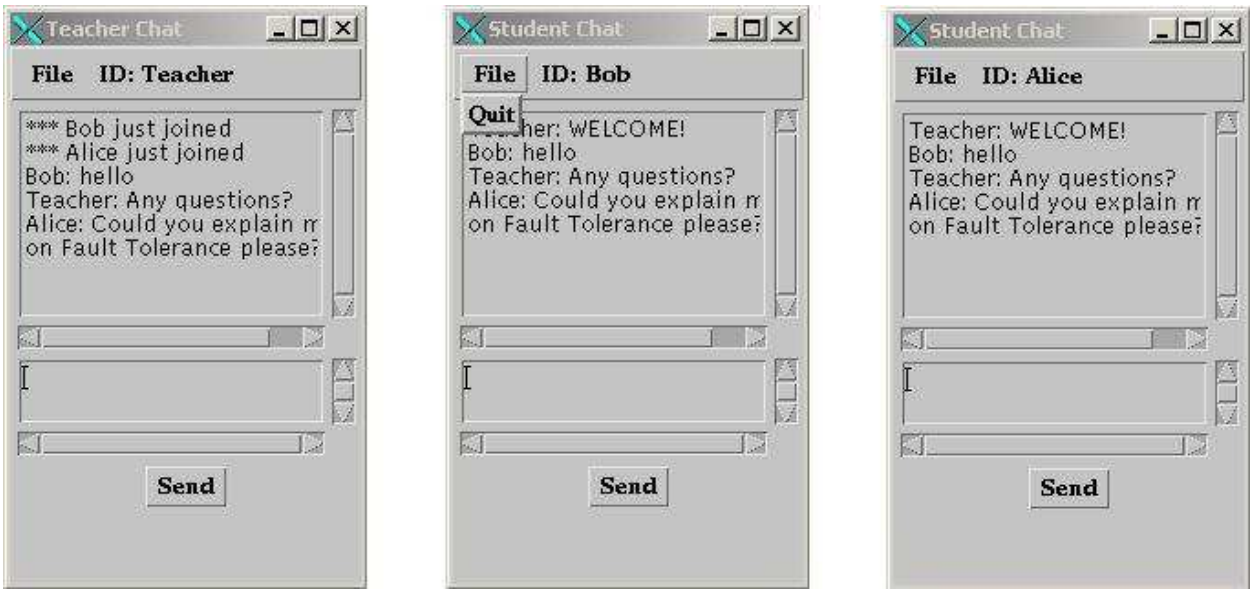


Figure 6.4: Sending and receiving messages during the chat session

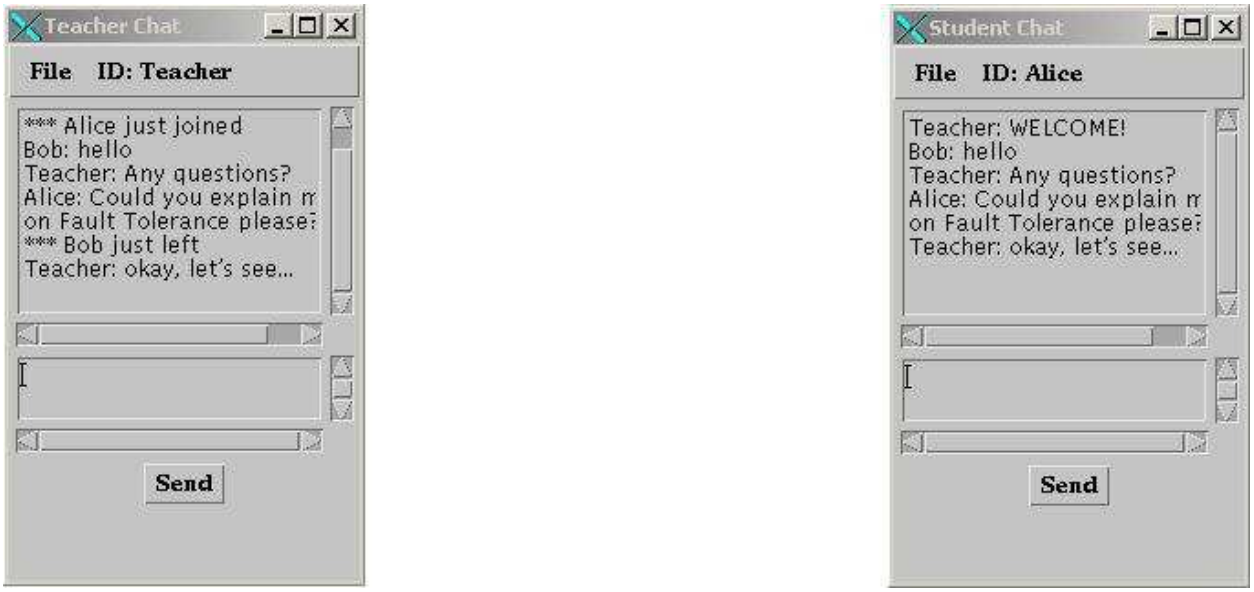


Figure 6.5: One student agent left the session, but the chat continues

This system works well and the idea of using a tuple space enables the system to handle loss of connection from the PDAs. When these PDAs manage to re-establish the connection, their agents automatically retrieve all messages sent to the tuple space while they were disconnected, hence no communication should be lost.

6.4. Setting the demonstrations

Ambient Campus is the only case study within the RODIN project that aims to provide (a set of) demonstrations. At this stage, we are planning a simple demonstration to be shown during the 4th Plenary Workshop in April 2006.

6.5. Future development

It is our plan to complete the development of the first prototype of the system supporting lecture scenario by summer 2006. Upon this completion, we will test the system, evaluate its development process and feedback our experience to work conducted in WP2-WP4. During the coming 8 months we will finalise our decision about the next scenarios. The lecture scenario could be followed by one of the following Ambient Campus scenarios: a library-based scenario or a medical school-based scenario. The library scenario places mobile devices into the core of activities performed in libraries. The system to be developed should enable library users to search and locate library materials on the move, to scan texts or images directly to their mobile device, to transfer collected data to their desktop computers, and to directly search and download materials from the web as they found a reference to it in a library material. The medical scenario would involve the Medical Education School of the University of Newcastle upon Tyne which plans to equip all first year students of the Medical Faculty with PDAs from 2006. With the help of these PDAs, the students will be able to perform a number of education activities when they are located in one of the university campuses. We are particularly interested in supporting distributed work of groups of students on joint presentations, essays, course work as well as in supporting discussion groups and questions and answers sessions which follow lectures and involve both lectures and students.

There is also a possible link with Nokia case study on developing formal technique within MDA context (CS3). Later in the project we will analyse how to enrich the MITA framework with the abstractions and fault tolerance solutions we are developing for this case study and if this is possible, we will investigate further how the framework can be applied in developing this case study.

Another possible future development – the usefulness of which we will evaluate later in the project – is the design and implementation of the ambient campus system on other mobile platforms such as smart-phones and laptops.

6.6. Summary

This section outlined the progress made during the first year of the Ambient Campus case study. The major results include the completion of the requirements document; the development of the infrastructure for supporting Ambient Campus system using the B method; the exploratory research on the mobile technologies; and the implementation of a simple example demonstrating our ability to develop ambient applications.

There is still a lot of work to be done, including further refinement of the ambient infrastructure

and the implementation of a fully featured demonstration software system. Based on the progress that we have made so far, we expect that we will be able to meet the challenging objectives set for this case study in the remaining two years.

6.7. Acknowledgements

We are grateful to Jean-Raymond Abrial for his insightful comments on the initial version of the Ambient Campus requirements document.

6.8. References

[6.1] Rigorous Open Development Environment for Complex Systems (RODIN), Description of Work, IST 6th Framework Programme, Proposal No. 511599, April 2004.
[6.2] RODIN Deliverable D4 – Traceable Requirements Document for Case Studies, Project IST-5111599, February 2005.
[6.3] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
[6.4] Atelier B, available from http://www.atelierb.societe.com/index_uk.html.
[6.5] A. Iliasov, V. Khomenko, M. Koutny and A. Romanovsky. On Specification and Verification of Location-based Fault Tolerant Mobile Systems. *In Proc. REFT'05 - Workshop on Rigorous Engineering of Fault Tolerant Systems*, Newcastle upon Tyne (UK), Technical report, University of Newcastle upon Tyne, UK, July 2005.
[6.6] R. De Nicola, G. Ferrari, R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315-330, IEEE Computer Society, 1998.
[6.7] G. P. Picco, A. L. Murphy, G.-C. Roman. Lime: Linda Meets Mobility. *Proc of the 21st Int. Conference on Software Engineering (ICSE'99)*, Los Angeles (USA), May 1999.
[6.8] G. Cabri, L. Leonardi, F. Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing*, 4(4):26-35, July 2000.
[6.9] Ewe programming system, available from http://www.ewesoft.com.
[6.10] IBM's J9, available from http://www.handango.com.
[6.11] CrEme, available from http://www.nsicom.com.