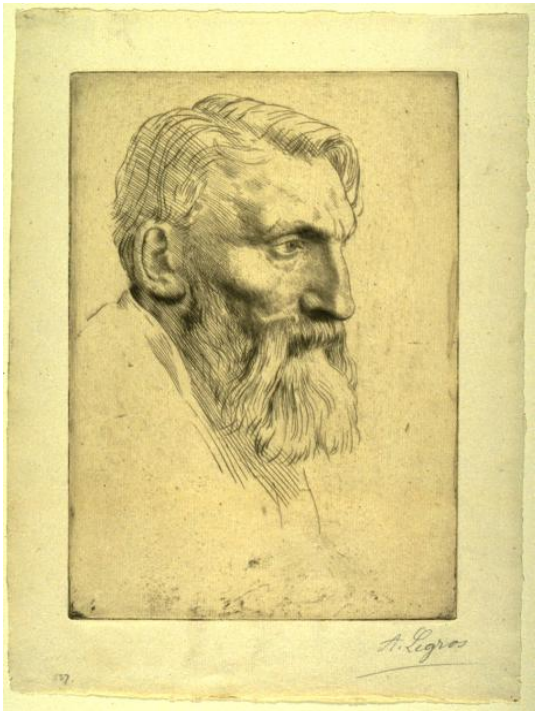
Project IST-511599
RODIN
"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable D27

Case Study Demonstrators

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

Public Document

30th October 2007

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

Contributors:

Budi Arief (University of Newcastle upon Tyne, UK),
Michael Butler(University of Southampton, UK),
Alex Iliasov (University of Newcastle upon Tyne, UK),
Dubravka Ilic (Aabo Akademi University, Finland),
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),
Qaisar Malik (Aabo Akademi University, Finland),
Mats Neovius (Aabo Akademi University, Finland),
Ian Oliver (Nokia, Finland),
Mike Poppleton (University of Southampton, UK),
Abdolbaghi Rezazadeh (University of Southampton, UK),
Alexander Romanovsky (University of Newcastle upon Tyne, UK),
Kaisa Sere (Aabo Akademi University, Finland),
Colin Snook (University of Southampton, UK),
Jenny Sorge(University of Southampton, UK),
Elena Troubitsyna (Aabo Akademi University, Finland)

CONTENTS

SECTION 1. INTRODUCTION

This document overviews the demonstrators of case studies. The aim of producing the demonstrators is to evaluate the impact of RODIN on the corresponding domains. The main body of this deliverable is the set of demonstrators uploaded to sourceforge at RODIN section. The demonstrators are the formal specifications, reusable abstract patterns, guidelines, examples of tools use, executable software. Their greatest practical value is that they can be freely downloaded and reused by anyone who is interested in application of RODIN methods and tools. The demonstrators in the form of formal specifications offer the examples of formal models which can be fed into the platform and analyzed. This promotes "learning by examples". The demonstrators in the form of guidelines summarize the experience of application of RODIN methodology and offer a strategy for developing similar systems from certain application domain. The examples of tool use demonstrate the benefits of various RODIN tools. Finally executable software offer tools that can be used to automate formal development and verification. This deliverable merely serves as the introduction to the demonstrators available from the web. It contains a brief overview of the demonstrators available for each case study together with the guidance on how to use the demonstrators.

In Section 2 we overview the demonstrators of case study 1 – *Formal Approaches to Protocol Engineering*. The case study investigates the use of formal methods in model-driven development of communicating systems and communication protocols. The first demonstrator for this case study is the Lyra UML profile, and formal specifications modelling intra- and inter-consistency conditions required for checking syntactic and structural consistency of Lyra UML models. The second demonstrator is an example of automatic translation of UML models into B specifications using the ATL tool. The third demonstrator is a large collection of Event-B models for verification of service decomposition and distribution phases of Lyra and reasoning about fault tolerance. We also present a formal modelling of parallelism in service execution. Finally, we present examples of test generation using the developed model-based testing methodology.

Section 3 describes the demonstrators of case study 2 – *Engine Failure Management System (FMS)*. The use of the RODIN technology is expected to improve accuracy of modelling the domain in order to reduce the semantic gap that exists between application requirement and system design and promote re-useability by being able to develop configurable generic model where other engine variants requiring FMS systems could be catered for. The demonstrator of case study 2 consists of four parts. The first part is the animation of compete FMS model, which demonstrates how the domain can be accurately modeled. The second part encompasses examples of genericity and re-use in the development. The third part demonstrates the take-up of the development methodology presented in D26 with the support of UML-B and RODIN. Finally there is a small collection of examples identifying RODIN methodology pitfalls, which points out the future research directions.

In Section 4 we overview the demonstrators for case study 3 – *Formal Techniques within MDA Context.* The demonstrators for this case study come in the form of technical papers and internal NOKIA reports. The demonstrators address the problems of making UML modeling more formal and adding rigor to model-driven development. The demonstrators describe the experience on how to make verification of system properties easier and friendlier for the engineers. Finally, we point out the work of building extendable development environment.

In Section 5 we give an overview of demonstrators of case study 4 – *CDIS Air Traffic Control Display System.* The major problem spotted in the CDIS development a decade ago was a poor comprehensiveness of the formal specification, lack of any mechanical proof of the specification consistency and continuity from the specification to design. To tackle these problems in RODIN we conducted an experiment – development of a "vertical" slice of CDIS system in the Event-B formalism. The results of this experiment constitute the demonstrator for case study 4. Our experiment demonstrates that the identified issues can be tackled by using refinement to layer in the functionality of the system in series of steps. This incremental approach modularises the proofs into small steps making the proof effort amenable to automated proof. Such an incremental approach also improves the comprehensibility of the formal specification which is important for maintenance and evolution of the specification.

Finally, in section 6 we overview the demonstrators of case study 5 – *Ambient Campus*. The aim of this case study is to investigate the use of formal methods combined with advanced fault tolerance techniques in developing highly dependable ambient intelligence applications. The demonstrator shows the entire process of the development of the Ambient Campus – from informal requirements to formal specification and checking dynamic properties by Mobility Checker. The development is illustrated by the screen-shots demonstrating tool use.

Let us note, that the participants of RODIN Industry Day which was held in Paris, on September 10th, have expressed a strong interest in formal models produced in the case study developments. Hence we believe that the demonstrators will serve a mechanism for disseminating RODIN methodology and help to ensure continuity of the project after its completion.

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

2.1 Introduction

This section of the D27 report presents the demonstrators produced in case study 1. The domain of the case study is model-driven development of telecommunication systems and communicating protocols. In particular, we focus on the service-oriented approach Lyra developed at the Nokia Research Center. The approach is based on decomposition / composition techniques. It uses UML2 as its model language.

Currently algorithmic verification is used to verify the correctness of the decomposition steps. However, telecommunication systems tends to be very large and data intensive so that the use of model checking is prone to the state explosion problem. Therefore, the important research area is to investigate how formal top-down (refinement) techniques can help to verify the Lyra development process.

Another important question is applicability of techniques for formal reasoning about fault tolerance in the development of telecommunication systems. In particular, we have investigated how the fault tolerance mechanisms can be incorporated and verified within the formalised Lyra development process.

2.2 Demonstrators of RODIN advances

Before presenting the demonstrators for this case study, we give a brief overview how the problems mentioned in the introduction have been tackled during the project.

In order to evaluate feasibility of using formal refinement techniques to verify model decomposition, we have formalised the Lyra development process into the corresponding specification and refinement process of Event-B. The B refinement has been used to demonstrate the correctness of Lyra decomposition steps as well as incorporation of fault tolerance mechanisms. This has been achieved by developing B specification and refinement patterns reflecting essential Lyra models and transformations.

Moreover, to make our support for the Lyra approach more complete, we have created additional automation steps resulting in a tool chain for Lyra-B integration (see **Fig.2.1**). In particular, the Lyra UML models to be automatically translated into Event-B have be checked for syntactic and structural consistency. This is done using the Lyra UML profile and additional consistency conditions developed during the project. Also, the results of the formalised Lyra development are used as inputs for automatic test

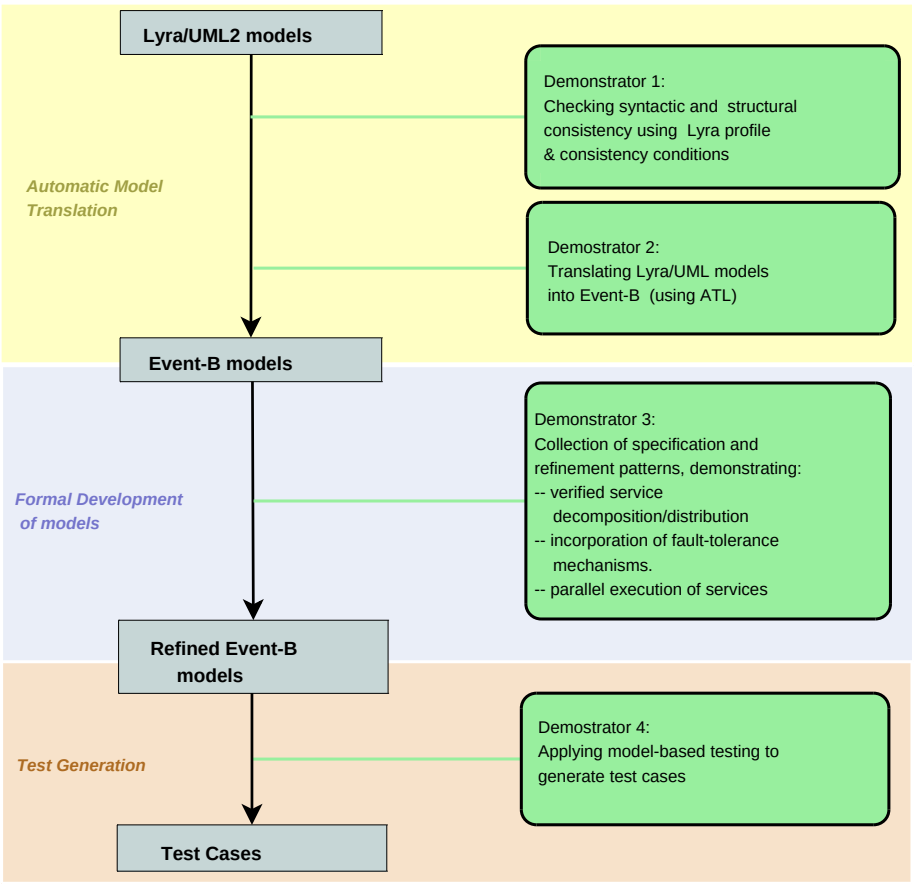


Fig. 2.1: Tool chain for Lyra-B integration and demonstrators

generation using the created model-based testing methodology.

The demonstrators presented in this section illustrate our results at different points of the automated Lyra design flow (**Fig.2.1**). Their full list is as follows:

1. The Lyra UML profile, intra- and inter-consistency conditions for checking syntactic and structural consistency of Lyra UML models;
2. An example of automatic translation of UML models into B specifications using the ATL tool;
3. The collection of Event-B models demonstrating
   - verification of service decomposition and distribution phases of Lyra,
   - incorporation of fault tolerance mechanisms,
   - modelling parallel execution of services;

4. An example of test generation using the developed model-based testing methodology.

Now we will describe these demonstrators separately in more detail. All the demonstrators will be accessible from the Rodin section of the sourceforge site.

2.2.1 Checking consistency of Lyra UML models

To automate the Lyra design flow, we need to know the precise form and structure of Lyra UML models that are used as inputs for our tool chain. For this purpose, a Lyra UML profile has been derived. The profile defines the Lyra-specific modelling concepts and dependencies between them, thus outlining the required stages of the system development. The profile is considered to be a reference model using which we could validate created Lyra models.

Also, our work allowed us to establish consistency between the Lyra UML2 models while undertaking the Lyra development, which otherwise we could not achieve within the profile solely. While verifying the Lyra development flow, we simulated Lyra development and formalized both the Lyra models and the intra- and inter-consistency rules in B.

The demonstrator will include the documents of the developed Lyra profile as well as formulated additional consistency conditions for Lyra UML models. Moreover, we will demonstrate the B models specifying the process of creating Lyra UL models in a consistent way. The ProB animator can be used to animate this verified model creation process.

2.2.2 Automatic translation of Lyra UML models

The Lyra profile and consistency conditions of Lyra models are used to direct automatic translation Lyra UML models into the corresponding Event-B specifications. The translation is accomplished by employing an external tool ATL based on Atlas Transformation Language.

The figure **Fig.2.2** provides an overview of the ATL transformation (Lyra2Event-B) that enables to generate an Event-B model, which is in fact an Event-B machine, conforming to the Event-B metamodel, from a Lyra model that conforms to the Lyra metamodel, which is in fact a Lyra Profile. The designed transformation, which is expressed by means of the ATL language, conforms to the ATL metamodel.

The figure **Fig.2.3** shows part of the Lyra profile in graphical format used for Lyra

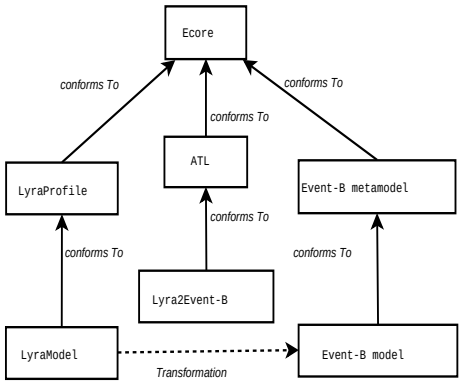


Fig. 2.2: An overview of Lyra to Event-B tranformation using ATL

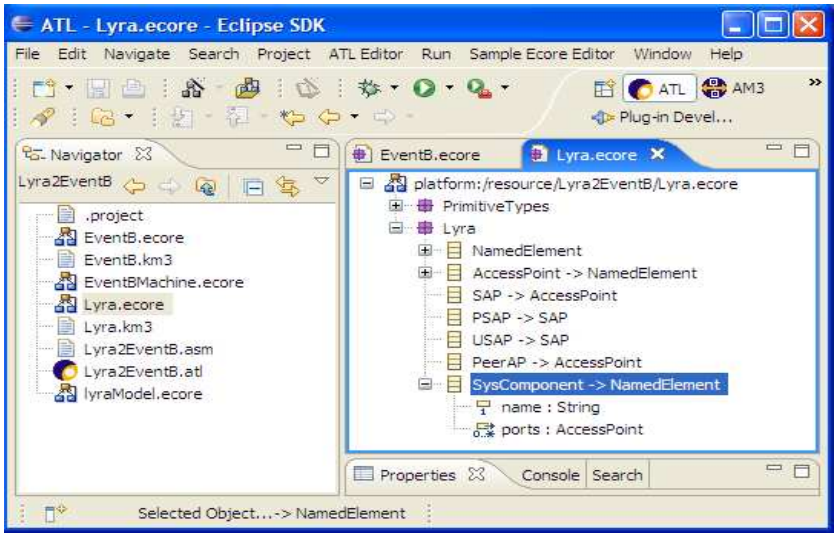


Fig. 2.3: Part of Lyra Profile used for trannsformation

model transformations. Lyra models, conforming to the Lyra profile, are transformed into corresponding Event-B machines, conforming to the Event-B metamodel. These transformations are directed by using special rules written in the ATL language. The rules define the exact way Lyra UML elements should be translated into the corresponding elements of Event-B.

The demonstrator will include an example of translation of a simple Lyra UML diagram into the corresponding Event-B specification according to the defined ATL transformation rules. Moreover, it will be demonstrated how the models violating the Lyra UML profile and / or the consistency rules are identified during the translation process.

2.2.3 The B specification and refinement patterns

To verify the Lyra development process, the essential Lyra UML models and transformations have been formalised as the corresponding Event-B specifications and refinement steps. At the beginning these models focused on verifying the Lyra decomposition and distribution phases. Later on, however, the Event-B specifications and refinements have been enhanced to incorporate fault tolerance mechanisms as well as modelling parallel execution of services.

The arising complexity of these formal models is handled by introducing abstract data structures modelling service decomposition, distribution, and fault tolerance aspects of the system. This makes these B models into specification and refinement patterns that have to be instantiated with concrete data during the actual development process.

The demonstrator will include the collection of the aforementioned B models. Moreover, the guidelines for data instantiation of these models will be given separately.

2.2.4 Model-based test generation

The B specifications of the formalised Lyra development can also be used as models from which we can generate test-cases for the corresponding implementations. This is often referred to as *model-based testing*.

Our work on model-based testing of Lyra B models is being implemented as a plug-in for the RODIN open-source platform. The model-based testing (MBT) plug-in is designed in such a way that it uses the ProB model checker plug-in in the background. The ProB plug-in is capable of generating execution traces of the models. It is also used to verify the satisfiability relations between scenarios and their respective models. In order to generate test-cases through the MBT plug-in, the user is first required to prove correctness of the model(s) using the RODIN platform. Additionally, the user has to provide testing scenario(s) for the most abstract specification model. The MBT plug-in uses the user-provided scenario(s) and the provided B models to generate test cases. The process can be then repeated for each refinement step.

The demonstrator will include an example of generating test cases for selected Lyra B models using the execution traces produced by the ProB plug-in.

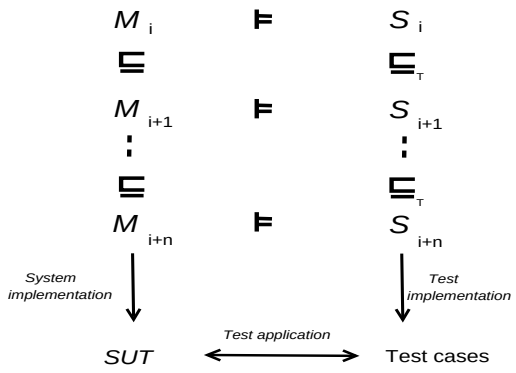


Fig. 2.4: Overview of the model-based testing approach

2.3 Conclusions

Our evaluation has shown that the work on the case study has progressed according to the initial plan and achieved the expected objectives. The achieved results allow integration of formal methods into the existing development process at Nokia through automation of the refinement steps in the design flow and automatic translation of Lyra/UML-2 models into the formal framework. Nokia considers the achievement of such automation as having added significant value to industrial system development.

SECTION 3. CASE STUDY 2: ENGINE FAILURE MANAGEMENT SYSTEM

3.1. Introduction

This section of the D27 report introduces the demonstrators produced in case study 2. In addition to the Engine Failure Management system case a second case production acceptance test "PAT" was undertaken in the final year. The PAT case is described in the D26 [3.1] deliverable as part of case study 2. Demonstrators from both domains have been provided to illustrate different case achievements.

3.1.1. The FMS Domain and motivations

The definition of the engine failure management system has been described in the D2 deliverable [3.6] and in the initial presentation of the project.

The FMS system is required to provide a dependable system by tolerating environmental faults with the following failure attributes

-Natural physical deteriation of hardware during operation
-Permanent or transient failures
- non malicious failures, deliberate or accidental
- 75% failures external to controller, 100% external to software

The case study domain aims are to achieve improvement in Failure managements subsystems maintenance and re-useability. The use of the Rodin technology is expected to contribute to improvement by;

1. Being able to accurately model the domain in order to reduce the semantic gap that exists between application requirement and system design. This is a problem for the domain developer that hinders FMS maintenance.
2. Promote re useability by being able to develop configurable generic model where other engine variants requiring FMS systems could be catered for.

3.1.2. The PAT Domain and motivations

The definition of the PAT system has been described in the D26 deliverable [3.1] as part of case study 2.

The PAT system is required to provide a dependable system to detect hardware failures in production units containing an FMS system.

The PAT system needs to be flexible to changing test requirements and has the case aim to be more generic.

This need for a more generic solution serves several purposes

1. Catered for instantiation of new test instances for other variants of units
2 Ease development of new test behaviour
3 Reduced the Validation time of the test system as fewer system components requiring verification

The system needs to integrate with existing test facilities so a further case aim is the consideration of apply formal methods involving a legacy implementation

3.2. Demonstrators of RODIN advances

3.2.1. FMS Demonstrators(University of Southampton)

There are four intended parts to the demonstrator of case study 2 from which a subset of possible demonstrators will be prepared.

(1) It will show the animation of the complete FMS model with some pointers to the more interesting parts of the model. *Addresses how the domain can be accurately modelled.*
(2) Genericity and re-use of the model will also be shown using suitable examples. *Addresses how reuse can be promoted in the domain.*
(3) The development methodology which was described in D26 [3.1] will be taken up and the viewer will be walked through this cycle step by step using one stage of the FMS development. *Demonstrates a methodical V&V approach to formal development with UML-B/RODIN*
(4) A small demonstration of different pitfalls of the RODIN platform and its plug-ins will be given. *Indicates areas for improvement and further development*

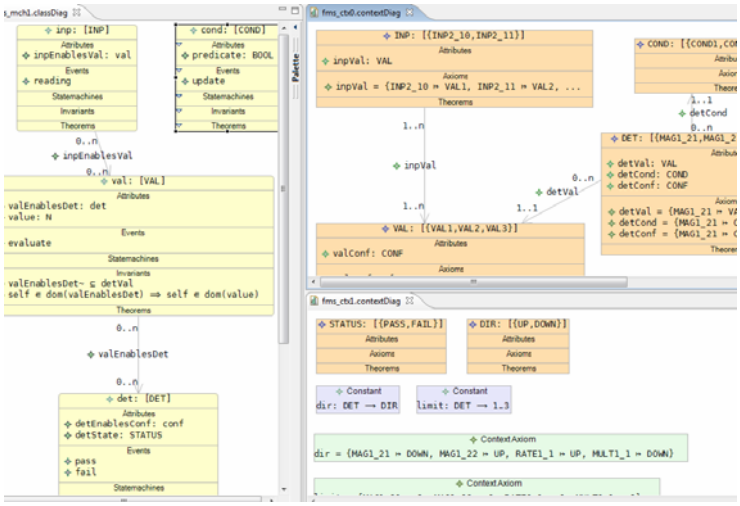
There are four main areas that will be considered for demonstration.

(1) FMS model animation

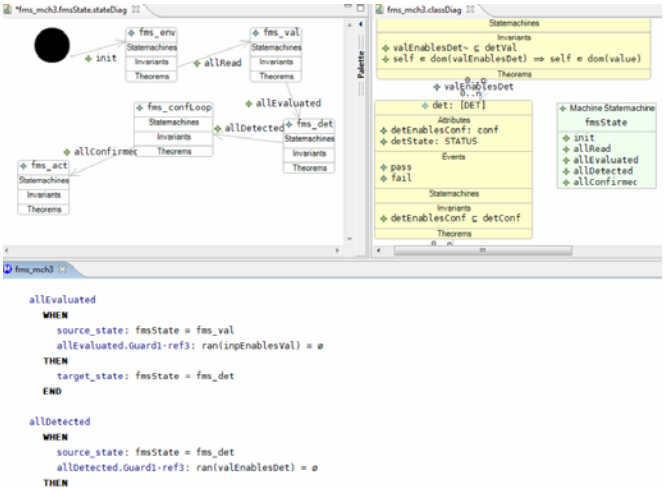
Firstly, the complete generic model with all its features will be shown. The model can be animated in Rodin up to refinement 1. The use of lambda expressions in refinement 2 prevents the animation of the model in Rodin because lambda expressions are currently not supported by the ProB animator plugin. The animation of refinement 1 in Rodin will be available as an annotated video. Further refinements will be made available as classical B machines, which can then be downloaded and animated using standalone ProB. A short video might be made available that demonstrates the history recording mechanism and the integrated system state.

- The UML-B model will be shown in class and context view (several tabs can be put next to each other) – like this we will demonstrate how different

models can be browsed and changed in one view.



- The generated Event-B will be shown for some features.
- We will then switch to the ProB perspective in order to animate the model.
- Further refinements cannot be shown in Rodin, because of the ProB translation error for the lambda expressions.
- UML-B of Refinement 2 history mechanisms will be shown with a link to the Event-B generation.
- The animation of history mechanisms will be shown in ProB and the machine file can be downloaded by the user.
- The final demonstration of the model will be the integration of Aabo's statemachine in refinement 3



- This will be done by browsing the model and showing the statemachine and the generated Event-B.
- A ProB animation will then demonstrate the working model.
- The B machine will be made available for download

(2) Genericity and re-use

Genericity and re-use will be shown by giving a demonstration of switching the context and an implementation of a different counting algorithm. Again, these will be shown in a short video and files will be made available as far as possible. We will generate manually a different instantiation of the context, which can then be switched with the existing one. Ideally, this should be done using the context

manager, but due to the immaturity of this tool, the demonstration will be made manually. There will be a short animation of the integration methodology used for integrating Aabo's ideas. This will mainly be a graphical animation with a quick demonstration of the model in the end. Again, this will be in the form of a video.

- We will demonstrate how to use the context manager.
- (if time allows) we will show how an existing counting algorithm can easily be integrated into the existing model – it will then be animated.

(3) V&V-oriented development methodology

The third part of the demonstrator will evolve around the development methodology, as described in D26. The steps involved are, briefly,

1. Animation (ProB plug-in) - validation
2.1 Model Checking (classical B, ProB) – validation
2.2 Model Checking (disprover, verify POs) - verification
3. Animation (ProB plug-in) – verification
4. Interactive Proof – verification

This V&V development cycle will be shown using one development stage of the FMS model. The viewer will be walked through the development of this refinement stage, which will then be validated and verified as described earlier, thus demonstrating the V&V methodology.

(4) Current errors and pitfalls

The last part of the demonstrator will be involved with the demonstration of some errors and pitfalls of the Rodin platform. This will be done by creating a video with annotations highlighting the kind of errors that may occur.

Rodin platform pitfalls

- Difficulties in using interactive prover.
- Some error messages are unclear.

UML-B pitfalls

- Refinement in UML-B is not yet mature, the machine to be refined has to be copied and pasted within the .umlb file. The refines clause has then to be set manually to indicate which events are refined.
- Generation of duplicate implicit context for refinement, which causes warnings

ProB pitfalls

- Lambda expression error will be shown. We will show the generated classical B and demonstrate how the file can be imported into ProB.
- Classical B translation without line breaks

Format of Demonstrators

All demonstrators will be accessible from the Rodin section of the sourceforge site.

Videos

The video demonstrators can simply be downloaded and played back using a media player. Those videos are a great way to quickly show certain features of the model –

they guarantee to bring across what should be shown and the user will have no problem navigating this video.

Downloadable Models
Some parts of the FMS model will be made available for download. The user can import them into the Rodin platform and animate them and assure that it is fully verified. The parts of the FMS model that cannot be animated using the Rodin platform will be supplied as a machine file, so that it can be animated and model checked using “standalone ProB”.

3.2.2. PAT Demonstrators

The domain aim “to provide a generic system for the PAT” was addressed by providing a system which could be easily configured to execute different tests. The system developed allows the creation of test instances through a dedicated generic editor, which is encoded into a serialisation of commands. An interpreter executes these commands in order to perform the acceptance test.
The generic editor was developed using structural modelling and and applying EMF technology. The generic editor demonstrator is described below;

The aim to use formal methods with a legacy implementation was achieved through partial modelling behaviour of the interpreter. The interpreter reuses some existing legacy code in its lower level functionality. The legacy behaviour was modelled alongside new behaviour in a partial model. The partial model demonstrator is described below.

The demonstrators and their development are outlined in D26 [3.1].

3.2.2.1 PAT Editor Demonstrator

The PAT editor demonstrator is provided in the Rodin demonstrator area of sourceforge. It will be in the form of a screenshot example or a flash animation. Installation instructions and execution instructions are provided.
The demonstration shows how a manual test requirement can be encoded into the Generic editor. This also illustrates the close mapping of the problem domain to the design (an aim of the FMS study).
The execution of the created test is then shown which demonstrates the requirement has been configured correctly.
The modification of a test instance and its execution is illustrated which demonstrates the generic system domain configurability to meet its generic design aim.

3.2.2.2 PAT Partial Specification Demonstrator

This consists of a simple model stored in the Rodin sourceforge demonstration area. Installation instructions and execution instructions are provided. The model is described in section 3.2 of D26 [3.1].

3.3. Overview of Demonstrator Achievements

The demonstrators provide an example of how the domain aims described above have been met. Achievements of the cases have been outlined in D26 [3.1]. Assesment of their contributions to Rodins methods and tools and evaluation metrics are given in D28 [3.3]and D34 [3.4].

3.3.1. FMS Demonstrators Practical Value

The generic model provides a template which will support various developments in the domain. The model partially implements the original FMS specification [D4], thus providing the basis for near-automatic production of the product line of airframe-specific systems, each defined by airframe-specific configuration data. This is *static* genericity, supported by the prototype Requirements Manager and Context Manager tools, particularly for large configuration datasets. Further, [3.1] shows how, by feature-oriented structuring, the model serves as a *dynamically* generic template for other, behaviourally distinct product lines in the FMS domain.

The V&V-oriented approach demonstrated provides an exemplar of methodical working with UML-B/RODIN. The presentation of the demos provide practical way of learning the technology through video and the use of real models to use.

3.3.2. PAT Demonstrator Practical Value

Editor

The demonstrators practical value is its flexibility to implement a test configuration for use with an interpreter. The generic editor has been used to configure the test requirement of a real production test in the domain. The flexibility provides a test user to easily update or select a new set of tests as required. The configurability is only constrained by the availability of test items in the structural model.

Partial Model

The demonstrators practical value is its use for exploration of future model development of the PAT. It is used to isolate the dependency of new functionality on existing legacy functionality, which can then be subject to formal verification.

3.4. References

[3.1] RODIN deliverable D26 : d1.5 Final Report on Case study Developments IST-5111599, September 2007.

[3.2] RODIN deliverable D27 : d1.6 Case study Demonstrators IST-5111599, September 2007.

[3.3] RODIN deliverable D28 : d1.7 Report on assessment of tools and methods IST-5111599, September 2007.

[3.4] RODIN deliverable D34 : D7.4 Assessment report IST-5111599, September 2007.

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

Section 4. Case study 3 -- Formal Techniques within an MDA Context

4.1 Introduction

This case study is concerned with the formalisation of various subsets of the MITA platform [MITA] (developed in Nokia within the NoTA - Network On Terminal Architecture - project) and, more generally, with the formalisation of the infrastructure and techniques to allow MDA to be used more formally.

The CS3 team considers the following to be the most acute problems in the domain of the case study:

- Making UML development more formal
- Making property verification easier and friendlier for the engineers
- Adding rigour into MDA
- Building a rigorous development environment which can be easily extended

4.2. Demonstrators of RODIN advances

Making UML development more formal. Application of the U2B plugin and the step-wise development method it supports in the development of some parts of the NoTA system has helped the team to in adding rigorous into this development. This plugin is a good compromise between the use of UML as the dominant modelling technique in Nokia and Event B. Report [O1] describes three experiments and the experience gained in the use of formal methods in a software engineering environment, that does not completely rely on the top-down stepwise development. These experiments were based on the Use Case/SDL Based Development, UML Based Development and the UML with Explicit Architecting.

Making property verification easier and friendlier for the engineers. ProB provides much necessary support for the default theorem proving and thus verification techniques already present in the Rodin tool. Use of ProB was extremely useful in validating verified models - verification removes certain error conditions and ensures that the model to be validated is "correct". Verification in the style of development used in MITA became a secondary concern with the focus more on establishing that the specification met the customer's demands rather than on establishing the adherence to certain properties. This fits in well with the style of development commonly seen in industry where constructing a model to investigate the properties of the system is not always feasible - ProB and the validation style of development in this sense provides a way of first constructing and demonstrating systems then discovering properties later. The use of ProB was particularly useful with regards to the initial work made with the B language. In use ProB is stable and reasonably fast. Scalability is always an issue but in the sizes of models presented to the tool, no problems regarding this have been seen.

Adding rigour into MDA. Within this case study a method was developed [B1] for introducing formal transformation of platform independent models (PIM) to platform specific models (PSM) in a model driven architecture (MDA) context. While fault

tolerance is not introduced in the PIM to make the models reusable for different platforms, the PSM often has to consider platform specific faults. A model transformation of the PIM in order to preserve refinement properties in the construction of the fault tolerant PSM is presented using Event B as a formal framework for the reasoning

.

Building a rigorous development environment, which can be easily extended. The Rodin development environment offers a useful solution to this problem. The CS3 team experiments with the number of plugins (ProB, U2B) demonstrated the usefulness of the choice of using the Eclipse technology. The most important example demonstrating this was the team's work on developing and integrating a new plugin [O2] for supporting circuit development with Event B and Bluespec: the plugin integration was straightforward.

4.3. Overview of achieved results

The CS3 team has found these results to be very useful. They have contributed to the improvement of the overall development methodology used by making it more rigorous and easier to apply. Some parts of the MiTA system and applications (e.g. protocols) have been used in this work to demonstrate the results achieved. By the very nature of this work the real systems model are confidential and cannot be shown outside of the consortium but a considerable amount of the open work (including samples of the formal Event B models) is described in D26, D18 and in the published papers referenced in D26..

References

[B1] P. Boström, M. Neovius, I. Oliver, M. Waldén. Formal Transformation of Platform Independent Models into Platform Specific Models. In Proceedings of the 7th International B Conference (B2007), Besançon, France, LNCS. 4355, pp. 186-200, January 2007. Springer-Verlag.

[MITA] Mobile Internet Technical Architecture. IT Press. 2002.

[O1] I. Oliver. Experiments and Experiences with UML and B. NRC-TR-2007-006. May 2007. Nokia Research. Finland.

[O2] I. Oliver. Circuit Development with Event B and Bluespec - RODIN Plugin Overview. Presented FDL'06 (Forum for specification and design languages). September 19-22, 2006. Darmstadt, Germany

Section 5. Case study 4 -- CDIS Air Traffic Control Display System

5.1. Introduction

This section describes the CDIS case study of the RODIN project. The RODIN tool support has been used to develop a formal development of CDIS. In order to keep the case study manageable in the context of the RODIN project, a subset of the original CDIS has been carefully chosen for redevelopment [1]. However, rather than focusing on individual aspects of CDIS, a `vertical slice' has been taken so that all of the interesting features of the system are covered (albeit in a lesser form).

The objective of the experiment (that we shall refer to as `the CDIS subset') is to derive a methodology for large scale formal development. Redeveloping an existing system also allows us to reflect on the lessons learned from the original development. Our aim in this section is to demonstrate how we have attempted to overcome the lack of comprehensibility and formal proof of the original CDIS development by adopting a methodology that makes use of available tool support in an effective way.

The initial CDIS specification is necessarily complicated and the core specification has been criticised for its complexity. This complexity and the bottom-up construction in VVSL force a level of specification that is too detailed to get an appreciation of the overall system behaviour. Too much complexity also precludes formal analysis. In order to reason about a specification formally, it is necessary to keep the level of detail as simple as possible. Otherwise mathematical proof becomes infeasible.

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

5.2. Demonstrators of RODIN Advances

In this section we outline our experiment in applying the Event-B formalism and the RODIN tool to the CDIS case study. Our approach is focused on finding an appropriate solution for the challenging aspects that we have pointed out in the previous section. Our proposed solution should address these issues:

- Lack of any mechanical proof of the consistency of the specification
- Difficulty of comprehending the original specification

- Lack of any formal connection between the idealised specification and realistic distributed design

Our experiment demonstrates that all of these issues can be tackled by using refinement to layer in the functionality of the system in series of steps rather than trying to model all the functionality in one large specification. This incremental approach modularises the proofs into small steps making the proof effort amenable to automated proof. Such an incremental approach also improves the comprehensibility of the formal specification which is important for maintenance and evolution of the specification.

In an effort to link the idealise view of the centralised specification to the realistic view of the design level we have developed two different set of models. In the first stage by starting from an idealised view of the system we have developed a centralised version of the system. This includes a specification and a number of refinement levels. These models have served as platform to investigate different aspects of modelling such as:

- To investigate the appropriate steps to produce a layered formal model of the CDIS as an example and any large system specification in general.
- To have a realistic view of associated proof obligation which in turn can assist the formal modelling process in the following ways:
    - Fine tuning the model to have simpler proof obligation and achieve higher level of automatic proof discharging
    - To devise appropriate strategies to handle interactive proofs.
- As a basis for comprehending and discussing the system

By obtaining valuable findings and experience from the centralised version we are in a much better position to develop a distributed version of formal models. These models are based on a more realistic view of the system.
We believe that the approach we have taken and the lessons learned can be applied to the construction of large formal specifications more generally.

5.2.1. Description of the Achieved Results - Centralised Version

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

Abstract Specification (*CDIS_Context* + *ABS_DISPLAY*)

The abstract specification for a generic system includes two parts. The static part, which defines the reference sets and constants, such as ***Pages***, ***Displays*** and other related attributes is named ***CDIS_Context.***
The second part, the dynamic part, is defining the variables, their related invariants, and operation. The variable database represents the stored data, and ***page_selections*** records the page number currently selected at a user position. The variable ***private_pages*** holds the page contents of a page prior to release. This is intended to model an editor's ability to construct new pages before they are made public. Finally, ***trq*** models the `timed release queue' that enables a new version of a page to be stored until a given time is reached, whereupon it is made public.

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

First Refinement – Introducing Page Layout History

This refinement is not introducing significant changes into the specification. In this level we have just introduced the history for page layout. When an existing page layout has been updated by the editors, the system keeps the previous page layout as one step history of changes. In this level no new context has been introduced.

Second Refinement – Adding time

The abstract specification omitted many of the features that characterise CDIS. However, this made it possible to give a broad overview of the system, including its state variables and operations, within a few pages. Now we use the specification as a basis for further refinements in which the omitted details are introduced. As a second refinement, we introduce a notion of time so that we can add age information to attributes, and add creation and release times to pages. This will give us the necessary apparatus to model the intended behaviour of the timed release queue. This refinement and subsequent refinements will demonstrate how important features of CDIS are added to the specification incrementally.

In terms of the CDIS subset, there are two main reasons for adding time: each piece of airport data has an age which affects how it is displayed, and the version of each page that is displayed is also time-dependent. In this refinement we shall once again use our proposed syntax for record types [2]. In this stage both the context and the model have been extended with appropriate constructs to deal with the notion of time.

Third Refinement – Introducing Critical Fields and Acknowledge

Several other aspects of CDIS can affect the way values are displayed. One requirement is that there is some critical information which they subjected to regular updates. Any

new updated values should be highlighted when they are displayed and they should be acknowledged by the operators. Hence, with each attribute value, we need to record whether it is a critical field or not. When a critical field has been updated it may effects many active pages currently viewed by different user positions. In this stage we have introduced the ***edd_acks_required*** and related sets and constants to extent our model with the critical fields' requirement. Again here we have extended both the context and the model.

Fourth Refinement – Introducing Page Overlays

Airport pages comprise a pair of graphic background overlays and a layout descriptor for transient data fields. One overlay is permanently displayed, the other is selectable using the reveal/conceal facility. Transient data fields are always displayed – they are unaffected by the reveal/conceal state. Airport pages need to be validated to ensure that none of the transient data fields are obscured by the background or overlay.

The reveal/conceal facility applies only to ***EDDs*** displaying pages. For all ***EDDs*** there is a means of toggling the reveal/conceal state of the display. This affects only those pages on display that consist of a permanent background and an overlay. Concealed displays models the set of ***EDDs*** for which the overlay is concealed. The overlays of pages displayed on all other **EDDs** are revealed. The act of toggling reveal/conceal at an **EDD** adds the ***EDD*** to the set if it is not a member beforehand, otherwise removes it. The specification has been augmented by a fourth refinement in which this features are introduced.

Fifth Refinement – Highlighting Manual Interaction

Another aspect of CDIS that can affect the way values are displayed is manually updated values. One requirement is that any manually updated values should be highlighted when they are displayed. Hence, with each attribute value, we need to record whether it was updated manually. Once again, we use our notion of record refinement to achieve this.

The Boolean value associated with the new field manually updated indicates whether the attribute's latest recorded value (accessed via the value field) has been input manually. In this case, we extend the record type ***Attrs*** with a Boolean flag which indicate whether or not the field has been updated manually. We included this requirement in the fifth refinement level.

Sixth Refinement – Introducing Concrete Values and Error Handling

The ultimate aim of the refinement process is to construct a specification in which constants and variables are associated with concrete values and events are defined to maintain the state accordingly. As part of this process, we have to separate a single abstract type into several subtypes. In the case of CDIS, this technique is used to introduce concrete attribute identifiers and value types into the specification. For

example, the original VDM specification defines ***Attr*** value as a union type made up of value types such as ***Wind_direction*** and ***Wind_speed***. Although union types do not exist in B, we employ the separation technique to achieve the same goal. We define a new context in which ***Wind_direction*** and ***Wind_speed*** are defined as subtypes of ***Attr_value***.

Wind_direction and ***Wind_speed*** are just two examples of many different specialised values for the ***Attr_value.*** In the two subsequent refinements we have introduced many other examples of similar cases. From these refinements, it is necessary to amend the update event to ensure that only values of the correct type update the database. Previously, ***SET_DATA_VALUE*** updated any attribute identifier with any attribute value. Now it must be refined in such a way to avoid having a collection of events, each referring to specific attribute identifiers and attribute values. Again here by the use of Constant mapping we have introduced a matching function to eliminate this barrier.

5.2.2. Description of the Achieved Results - Distributed Version

As it has been stated earlier, we have attempted to construct a more realistic specification which unlike initial VVL specification can be liked to the distributed design. Using the experiences that we have achieved by developing the idealised version, we have developed a set on new models which includes both horizontal and vertical refinements. As usual these models include one specification and few refinement levels.

Abstract Specification for Distributed Version of CDIS

This specification is an extension of the idealised version. We have introduced a history tracking system both for the transient data and the page layouts and selections. In a realistic system it is possible that different terminals in different user position have a different view of both transient data and page layout in a specific time. These differences arise due to delays in the system.

In our distributed specification we have assumed that we can model the actual behaviour of the system by storing the track of all applied changes over the time. By have the past history of all changes it is properly possible to model a system which allows that different terminals to have different view of the system data.
As an effect of introducing the history tracking mechanism our specification became noticeably more complex in comparison with the idealised version. Despite of this the development of this specification by extending the centralised version was a straightforward process.

Refinements of Distributed CDIS

After devising this specification of the CDIS which allow tackling issues that arise in a distributed system, there are two methods to refine the initial specification. One possible approach is to apply all subsequent refinement of the idealised version before applying any vertical refinement. An expected advantage of this approach is the similarities between the generated proof obligations. This can assist the developers to discharge the

interactive proof obligation more easily. After completing all horizontal refinements in which we introduce new requirements we can proceed to the vertical refinement. During this stage we have replaced the history tracking mechanism by a system which comprises a central database, a history of only changed data and a local database for each viewing position. The main advantage of this approach for implementation viewpoint is that storing the history of applied changes should need much smaller memory in comparison of storing the whole database history.

Another approach to the refinement of the distributed specification is that the vertical refinement precedes horizontal refinements. You have not pursued this path because we believe that the vertical refinement is a design decision and we should not mix specification with design decisions.

5.3. Overview of Achieved Results

A key factor in our success was the construction of good initial abstractions capturing the essentials of the system concerned. Such a skill is not easily transferable of course, but by providing good examples, such as the one here, we can help others understand how to construct good abstractions.

Comparing styles

A very different methodology and modelling style was adopted in the Event-B development than in the original VVSL development (VVSL is a variant of VDM). The original VVSL development produced a single large specification that was difficult to comprehend and impossible to reason about using the technology available at the time. In the Event-B development of CDIS we have focused our effort on tackling the comprehensibility issue and the issue of mechanical proof. We quickly found that both these issues could be tackled by using refinement to layer in the functionality of the system in series of steps rather than trying to model all the functionality in one large specification.

The layered development helped the comprehensibility considerably because we were able to capture the essential functionality of the system in the abstract specification. The abstract model is just under 4 pages of Event-B and we claim that this abstract model allows the reader to quickly grasp the essence of the system. Four subsequent refinements were used to introduce additional features of the system. The nature of these refinements was that they added additional details to the information structures and placed further constraints on when various actions could happen. The layered nature of their introduction means they can be absorbed in a stepwise fashion thus easing comprehensibility.

Effort

The main Event-B development represents about 6 months of effort. This includes time includes time spent learning to use the new tool efficiently as well as time spent checking the models and performing the proofs using the new interactive prover . It also includes

time spent on identifying and dealing with bugs and incompleteness in the tool, half way through the development and revising the development as a result. It is therefore difficult to compare directly with the time taken for the original VVSL specification. Our subjective assessment is that the time taken is comparable, with the advantage that in the case of the Event-B the development has been machine checked and proved.

Language and style

The mathematical language of Event-B and VVSL are equally expressive. The key difference was not the notation; rather it was the style of specification used in the Event-B development, in particular the use of refinement to layer in details of the functionality, which led to a more comprehensible specification. The layered approach, along with the new powerful RODIN tool, made it possible to mechanically check and prove the models. It is worth emphasising that the CDIS specification is necessarily complicated. Even though the core specification has been criticised for its complexity, it is unrealistic to expect any significant improvements in the size of a specification that captures all aspects of CDIS, regardless of the notation used. However, the bottom-up construction in VVSL forces a level of specification that is too detailed to get an appreciation of the overall system behaviour.

Mechanical Proof

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

All proofs were carried out using the RODIN tool and we found RODIN to be a powerful prover. The layered development eased the proof of consistency of the specification since at each step we had a small number of relatively simple proof obligations. In addition to the consistency proofs the RODIN tool now is capable of handling wider forms of proof such as Well-Definedness. All of these increase the confident in to the produced system.

Records and refinement

Beside this, we have provided a number of concrete techniques which are transferable to the construction of other large formal specifications. In particular we made strong use of the developmental pattern of extending records to add additional information to information structures and to extend function signatures in refinement steps. We identified and made use of a related pattern of wrapping abstract types within record structures in a refinement step, providing a standard pattern for a gluing invariant. We also made use of record sub-typing and record extension to differentiate structures in refinements and to add attributes to abstract deferred sets. These techniques allow us to

avoid unnecessary clutter at the more abstract levels. The techniques are easily supported by existing B provers and our experience is that the associated proof obligations are mostly automatically discharged.

Linking Between Specification and Design

One weakness of the initial development of the CDIS was that there was no formal link between the idealised specification and the actual distributed design. We have successfully overcome this problem by extending the idealised version to amore realistic specification. We have demonstrated that this specification could be refined to a distributed model with a reasonable effort in the context of the RODIN tool.

From the above mentioned points we can conclude that the reconstruction of the CDIS case study represents a methodological contribution to the construction of large formal specifications. Our experience shows that incremental construction through iterative refinement makes it feasible to apply tool-based formal analysis to large specifications. This increases our confidence in the specification greatly and provides the basis for tool-based formal development of a design and implementation. We also believe that this approach makes a large formal specification more accessible and comprehensible both to those constructing the specification and to others. We believe that the approach we have taken and the lessons learned can be applied to the construction of large formal specifications more generally.

References

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

[2] N. Evans and M. Butler: Proposal for Records in B, accepted for publication, FM06.

SECTION6. AMBIENT CAMPUS DEMONSTRATORS

6.1 Introduction

Mobile agent systems (MAS) are complex distributed systems made of asynchronously communicating mobile autonomous components. Such systems have a number of advantages over traditional distributed systems, including: ease of deployment, low maintenance cost, scalability, autonomous reconfiguration and effective use of infrastructure. MAS are distinct enough to require specialised software engineering techniques. A number of methodologies, frameworks and middleware systems were proposed to support rapid development of MAS applications [1, 2, 3, 4]. However, there is as yet no single widely recognised standard and the problem of building large and dependable MAS remains open. In addition to that, a major stumbling block in this key area of computer system design is the the complexity of verification of MAS. One way of coping with the complexity problem is to use formal methods supported by computer aided verification tools. We are using the Event-B version [5] of the B Method [6] to model our system as a system of communicating agents.

6.2 Demonstrators of RODIN advances

The final demonstrators of the Ambient Campus Case Study are related to the project work on the Student Induction scenario (See D26). They include

- the requirements document
- B and mobility models of the chosen Ambient scenario
- screenshots of the platform and plugin use
- a demonstration of the use of the developed scenario.

6.3 Requirements document

This section contains requirements to a system implementing the discussed scenario.

6.3.1 Requirements Taxonomy

We split the system requirements into the following five classes:

ENV	Facts about the operating environment of the system.
DES	Early design decisions captured as requirements.
FUN	Requirements to the system functionality.
OPR	Requirements to the system behaviour.
SEC	Requirements related to the security properties of the system.

6.3.2 Top-Level Requirements

First we attempt a very rough description of the system. The description captures different aspects of the system: environment, some design decisions, dictated by the motivation for this case study, and few general functionality and security requirements.

The purpose of the system is to help fresh students arriving at a university campus with the registration process.

FUN1	*The system helps fresh students to go through the registration process.*

The system is made of a traditional university campus, virtual campus and ambients.

DES1	*The system is made of university campus, virtual campus and ambients.*

Automated registration complements manual registration process.

OPR1	*A student must have a choice between automated and manual registration.*

If the system breaks down it should be possible to switch to the manual registration process.

OPR2	*Malfunctioning or failure of the automated registration support should not prevent a student from manual registration.*

The system must be reasonably safe to use. There should be no leakage of sensitive information about students and registration process.

SEC1	*The system should not disclose sensitive information about students.*

It should be hard to inject malicious software into the system.

SEC2	*The system must prevent malicious or unauthorised software to disguise itself as acting on behalf of a student or an employee.*

6.3.3 University

University campus forms the environment for the software-based registration process. The university campus is obviously not something that can be designed and implemented. However it is important to consider it in the development of the scenario as it provides an operating environment for other two parts which can be realised in software and hardware. Analysis of this environment also leads to many important requirements.

ENV1	*In university campus students interact with university employees.*

ENV2	*Students can freely move around while employees do not change their position.*

ENV3	*Each university employee is permanently associated with a unique location.*

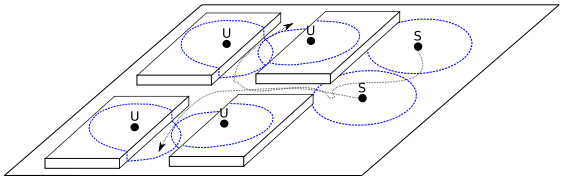


Figure 6.1: University campus is modelled as a number of university employees (U) and students (S). Virtual campus has the same structure but is populated with student and university agents.

6.3.4 Virtual Campus

Virtual campus uses software-based solution to process student registration automatically. Its organisation is similar to those of a real campus.

DES2	*Virtual campus is made of university agents and student agents.*

DES3	*In virtual campus, student agents can autonomously change their location.*

DES4	*Each university agent is permanently associated with a unique location.*

Virtual campus is a meeting place for student agents and university agents. During registration, student agent talks to different university agents.

FUN2	*Student agents and university agents can exchange information related to the registration process.*

Some registration steps require intervention from a student.

OPR3	*A registration process may fail due to incapacity of a particular university agent to handle the registration.*

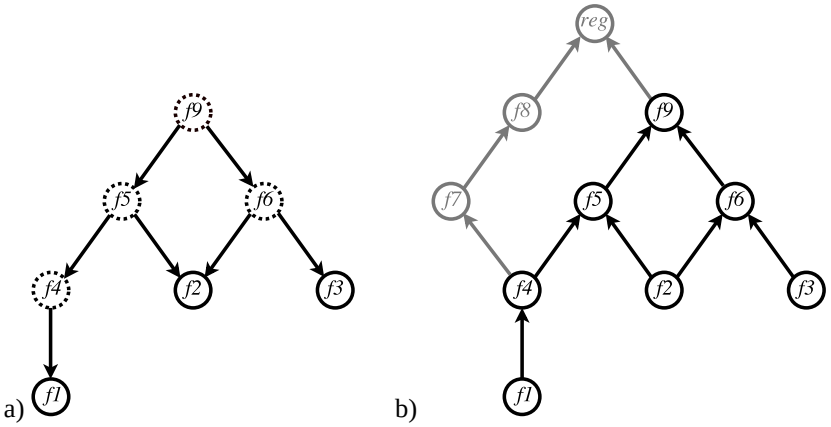


Figure 6.2: a) Registration process starts from a random location (*f9* on the figure). The basic requirements *f1*, *f2*, *f3* are discovered by tracing back the requirements graph. b) Student agent attempts to do the registration by satisfying each known requirement. It does not yet know the full set of registration requirements (unknown steps are greyed). They are discovered during this process.

Before some registration step can be attempted, a student agent has to go through other stages. This results in a tree of dependencies. The root of the tree is successfull registration and its leaves are the registration stages without any prerequisites. Student agent does not know about the tree structure and explores it dynamically.

OPR4	*Each registration stage has number of dependencies.*

DES5	*Initially, student agent does not know the dependency tree.*

DES6	*Student agent autonomously construct the dependency tree.*

Reconstructing the tree for each agent makes the system more flexible and robust.

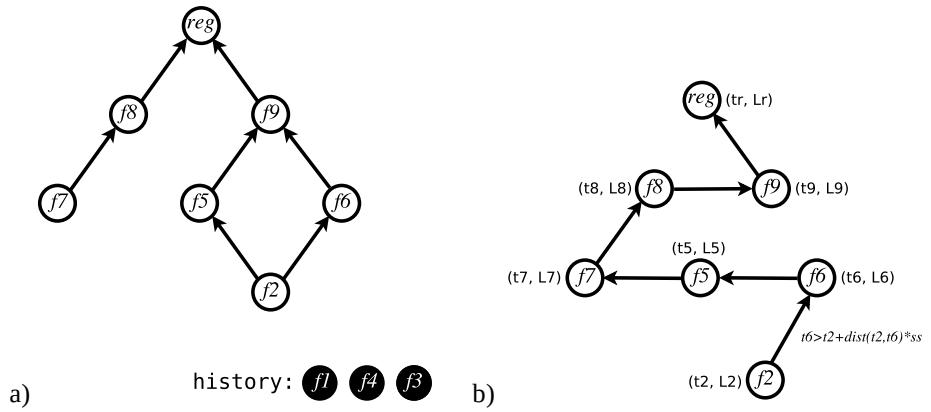


Figure 6.3: a) During registration a student agent accumulates registration information. b) Itenirary for manual continuation of a registration is a path covering all the remaining registration graph nodes and satisfying a number of constrains. A node of the path is described by a pair conatining when and where should go to resolve a given registration dependency.

Interacting with university agents, student agent records all the information related to the registration process. This information can be used to restraty registration or passed to the student to do manual registration. In the latter case, agent student creates and a schedule that helps a student to visit different university office in the right order and at the right time.

DES7	*Student agent keeps a history of registration process that can be used to restart registration from the point of last completed registration step.*

DES8	*Student agent can create an itenirary for a student to complete the registration manually.*

DES9	*Itenirary must satisfy the registration dependencies.*

6.3.5 Ambient

As implied by the scenario, ambients provide service within a predefined physical location. By service we understand an interaction of an ambient with student's software. Interaction is triggered when a student enters a location associated with a given ambient.

OPR5	*Ambients interact with student agents to assist with the registration process.*

FUN3	*Ambient provides services by interacting with student software.*

FUN4	*Interaction with an ambient is triggered when a student enters a location associated with the ambient.*

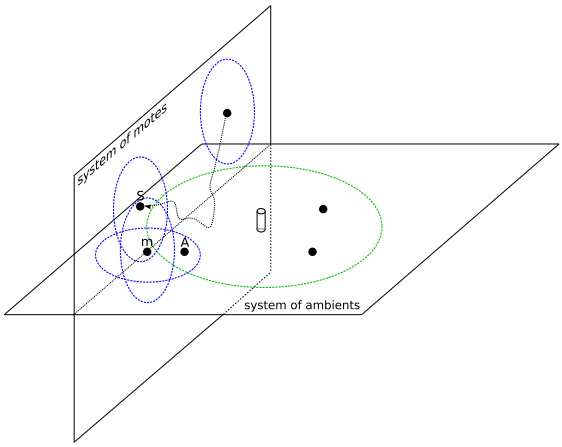


Figure 6.4: Composition of motes and ambients system.

When a student leaves an ambient location any interaction with the ambient must be terminated.

FUN5	*Interaction with ambient is terminated when a student leaves location of the ambient.*

Positioning Service

For simplicity, we assume that ambient locations are discreet - a student is either within a location or outside of it - and do not change over time.

FUN6	*Ambient locations are discreet and static.*

Discovery of an ambient by a student and a student by an ambient does not come for free. It is done using motes - tiny mobile sensor platforms**??**. A mote has low-power, short-range radio communication capability.

ENV4	*Ambients detect students nearby using the mote radio communication.*

Each student carries one such mote which broadcasts student identification at certain intervals.

ENV5	*Each student carries a mote.*

FUN7	*Student mote broadcasts student id.*

Student motes signals are sensed by ambients. Ambient agent is equipped with a mote radio receiver.

ENV6	*Each ambient is equipped with a mote radio receiver.*

When an ambient senses a student mote it transmits this information to all other ambients.

FUN8	*Position of a student detected by an ambient is made available to all other ambients.*

We will rely on this functionality to implement recovery in emergency situations.

6.3.6 Student

Automated registration must be under full control of a student. A student should be able to start, stop and inspect the current state of a registration.

FUN9	*Student starts and stops registration process.*

FUN10	*Student may inquiry for the current state of a registration while registration is in progress.*

FUN11	*When registration is finished or interrupted a student can access the recorded registration state.*

6.3.7 Student Agent

Student agent is a software unit assiting a student in registration.

FUN12	*Student agent assists a student in manual registration by creating a schedule for visiting university employees.*

FUN13	*Student agent records the state of registration process.*

6.3.8 Mobility

The scenario includes several types of mobility. There is physical mobility of computing platforms owned by students (e.g. mobile phones and PDAs). Students' agents can migrate to and from a virtual campus world. In this case agent code and agent states are transferred to a new platform using code mobility. Finally, agents migrate within a virtual campus using virtual mobility.

Different styles of mobility have different requirements. Code mobility is a complex and fail-prone process: it is dangerous to have an agent separated from its state or having an agent with only partially available state or code. There is also a danger of an agent disappearing during the migration: the source of migration, believing that migration was succesful, shuts down and removes the local agent copy, while the destination platform fails to initialise the agent due to transfer problems.

OPR6	*Agent either migrates fully to a new platforms or is informed about inability to migrate and continues at a current platform.*

Physical mobility presents the problems of spontaneous context change. A student agent may be involved into collaboration with an ambient when a student decides to walk away. Clearly, student behaviour cannot be restricted and such abrupt changes of context and disconnectios must be accounted for during design of agents and ambients.

OPR7	*Interaction between an ambient and student agent can be interrupted at any moment.*

Virtual mobility is the simpliest flavour of mobility as it does not involve any networking or anything actually moving in space. The only possible failure that can affect virtual migration is failure or shut-down of the hosting platform. However, such dramatic failure is unlikely to happen during an agent lifetime and thus we do not consider it at all in this document.

6.3.9 Security

There are a number of security issues with having an agent acting on behalf of a student. Agent with security-sesnsitive information, like information bank account and credit card numbers, must be protected from attacks of

malicious agents. There is an abundancy of security-related problems in agent systems. For this scenario, however, we focus on the following problems: authentication, man-in-the-middle attack and brainwashing.

To act on behalf of a student an agent must prove to its peers that it is authorised to do so. Similarly, a university agent must prove that It indeed acts on behalf of a university and is not a malicious agent phishing for students' personal information.

SEC3	*Student and university agents do mutual authentication before any interaction.*

Agent interaction must be securely isolated - there should be no possibility for other agents to intercept messages or insert new messages as this can be used to disguise a malicious as a valid agent acting on behalf of a student and also may give access to sensitive information, such as payment details.

SEC4	*Interaction of student and university agents must be private and securily isolated.*

A student agent in this scenario accumulates various information about the registration progress. A malicious agent may attempt to rewrite part of an agent state to mislead the student. This attack can happen not only in virtual campus but also during migration into or from a virtual campus.

SEC5	*Information gathered by a student agent must be protected from tampering.*

6.3.10 Fault-Tolerance

The system we are designing is complex distributed system with a multitude of possible failure sources. In additional to tradinational failures associated with networking we have to account for failures related to environmental changes which are beyond the control of out systems. Below is the list of faults we are going to address and which we believe covers all the important failures in our system:

- disconnections and lost messages:

OPR8	*Agents must tolerate disconnections and message loss.*

- failure of ambients:

OPR9	*Student agents must be able to autonomously recover from a terminal ambient failure.*

  also, since ambient services are not critical it si better to avoid failing or misbehaving ambient:

FUN14	*Student agent drops interaction with an ambient if it suspects that the ambient is malfunctioning.*

- failure of university agents. University agents are critical for completion of registration so it is worth trying to recover cooperatively:

OPR10	*Student and university agents cooperate to recover after failure.*

  It does not make sense to remain in virtual campus if one of university agents is failing to interact:

FUN15	*Student agent leaves virtual campus when it detects a failed university agent.*

- failure of student agents. Failure of a student agent may be dected by a university agent, ambience agent or student.

FUN16	*University agent detecting student agent carsh should attempt to notify the agent owner.*

And there is a possibility that a student suddenly terminates without leaving any notice. In this case we rely on student to detect this situation and possibly try again by sending another agent.

FUN17	*Student should be able to restart registration process.*

6.4 B and mobility models of the chosen Ambient scenario

To ensure interoperability of of different agent types in our scenario and also to verify properties such as eventual termination of the registration process, we use the combination of CSP process algebra, AgentB modelling - Event-B with some syntax sugar, and the Mobility plugin. The AgentB part of the design is responsible for modelling functional properties of the system, for the verfication purposes it is translatable into proper Event-B models. With the mobility plugin we able to construct scenarious describing typical system configurations and verify properties related to system dynamics and termination. For example, we can model check the migration algorithm described in Event-B to verify that the algorithm will never omit a location.

The whole development is lengthy, so we show only some excerpts.

Our system is concerned with registration of a fresh student. At a very abstract level the registration process is acomplished in one step

$$\frac{S_0 \xrightarrow{\text{REF\_PREFIX}} S_1 \text{ sat. FUN1}}{register.}$$

From the description of the system we know that the registration process is made of an automatic or manual parts, any which properly implements the registration process

$$\frac{S_1 \xrightarrow{\text{REF\_ICH}} S_2 \text{ sat. OPR1}}{\begin{array}{l|l} auto \mapsto register & auto. \sqcap manual. \\ manual \mapsto register & \end{array}}$$

(steps S_3 - S_6 omitted)

At thi stage we are ready to speak about roles of agents implementing the system. We introduce two roles: student (s), representing a human operator using a PDA and agent (a) which for now stands for all kinds of software in our system.

$$\frac{S_6 \xrightarrow{\text{REF\_ROLE}} S_7 \text{ sat. DES1}}{\begin{array}{l|l} s, a \in \rho S_7 & \begin{array}{l}(s'send \rightarrow a'move.; a'communicate.; a'automatic.) \sqcap (\\ \quad (auto\_fail \rightarrow manual\_anew.) \sqcap \\ \quad (auto\_part \rightarrow manual\_cont.))\end{array} \end{array}}$$

In the next model we focus on a submodel of the system which represents virtual campus activities, the $communicate.$. The process is refined into a loop where a student agent visits different university agents and speaks to them. The loop alterates between termination (**break**) and the registration process

$communicate.$ from $S_7 \xrightarrow{\text{REF\_LOOP}} S_1^{vc}$ sat. FUN3	
$done \mapsto communicate$	$a'^{+}(auto\_register. \sqcap \mathbf{break}); a'done.$

(steps S_2^{vc} - S_6^{vc} omitted)

Adding more details about interactions of student and university agents we arrive to the following model. The model implements a simple request-reply protocol where the university agent role is given a choice of a number of replies. Event $reply\_ok$ ok is used when registration is successfull, event $reply\_docs$ indicates that there are missing documents and that student agent must visit some other virtual offices before registration can be completed. In a case when registration is not possible without a student itself, the $reply\_pers$ reply is used.

$$S_6^{vc} \xrightarrow{\text{REF\_DCPL}} S_7^{vc}$$

$$sa'^{+}\begin{pmatrix} sa'migrate \rightarrow sa'ask.; (\\ \quad (ua'reply\_ok.; sa'save\_repl.)ua' \sqcap \\ \quad (ua'reply\_docs.; sa'doclist.)ua' \sqcap \\ \quad (ua'reply\_pers.; sa'do\_pers \rightarrow sa'\mathbf{break}))ua' \sqcap \\ \quad (ua'fail.; sa'leave\_vc \rightarrow sa'\mathbf{break})) \end{pmatrix}; sa'done.$$

(steps S_8^{vc} and S_9^{vc} omitted)

This model prepares to the transition to a state-based model with completely decoupled agent roles

$$S_9^{vc} \rightarrow S_{10}^{vc}$$

$$\begin{array}{l}
([\psi_1] \sqcap \mathbf{skip}) \| \\
^{+}(\psi_1 \rightarrow (sa'(migrate \rightarrow ask.); [\varphi_1])) \| \\
^{+}(\varphi_1 \rightarrow ua'((reply_ok; [\varphi_2]) \sqcap (reply_docs; [\varphi_3]) \sqcap (reply_pers; [\varphi_4])) \sqcap (fail; [\varphi_5]))) \| \\
^{+}(\varphi_2 \rightarrow sa'save_repl.; [\psi_1]) \| \\
^{+}(\varphi_3 \rightarrow sa'doclist.; [\psi_1]) \| \\
^{+}(\varphi_4 \rightarrow sa'do_pers.) \| \\
^{+}(\varphi_5 \rightarrow sa'leave_vc.)
\end{array}$$

The first version of a state-based model is a mere translation of the CSP expression

$S_9^{vc} \rightarrow S_{10}^{vc}$

$\psi_1 \mapsto migrate$
$\varphi_1 \mapsto reply$
$\varphi_2 \mapsto save$
$\varphi_3 \mapsto goaway$
$\varphi_4 \mapsto do\_pers$
$\varphi_5 \mapsto leave\_vc$

```
do(migrate).

role sa
  seq
    react migrate()
      skip
    act ask()
      say reply
  react save()
    say migrate
  react goaway()
    say migrate
  react do_pers()
    skip
  react leave_vc()
    skip

role ua
  react reply()
    case true refines reply_ok
      say save
    case true refines reply_deps
      say goaway
    case true refines reply_pers
      say do_pers
    case true refines fail
      say leave_vc
```

(steps S_{11}^{vc} - S_{13}^{vc} omitted)

A more detailed model features variables and action bodies specifying the protocol between student and university agents. At this stage we can make transition to a skeleton code implementing the specification

$$S_{13}^{vc} \rightarrow S_{14}^{vc}$$

do$(migrate : loc_0)$.

sets OFFICE, DOC
constants
  $deploc : DOC \rightarrowtail\!\!\!\!\!\!\rightarrow$ OFFICE
  $finreg$: DOC
properties
  $\bigcup_i mydoc_i =$ DOC
  $\bigcup_i myloc_i =$ OFFICE
  $\forall (i,j) \cdot (i \neq j \implies myloc_i \neq myloc_j)$
  $\forall i \cdot (mydoc_i \notin reqdoc)$

role sa
  variables $loc, docs, trace$
  invariant
    $loc \in$ OFFICE
    $doc \in \mathcal{P}$(OFFICE)
    $trace : \mathbb{N} \nrightarrow$ OFFICE
  initialisation
    $loc :\in$ OFFICE
    $docs := \varnothing$
    $trace := \varnothing$
  seq
    react $migrate(l)$
      $loc := l$
      say $reset()$
    act $ask()$
      say $reply(docs, loc)$
  react $save(newdoc \in$ DOC$)$
    when $trace = \varnothing \wedge l \in$ OFFICE $\setminus loc$
      $docs := docs \cup \{newdoc\}$
      say $migrate(l)$
    when $trace \neq \varnothing \wedge l = history(max(dom(trace)))$
      $trace := \{(max(dom(trace))\} \lhd trace$
      $docs := docs \cup \{newdoc\}$
      say $migrate(l)$
  react $goaway(ls \in \mathcal{P}$(OFFICE$))$
    when $l \in ls$
      $trace(max(dom(trace)) + 1) := loc$
      say $migrate(l)$
  react $do\_pers()$
    skip
  react $leave\_vc()$
    skip

role ua_i
  constants
    $myloc_i \in$ OFFICE
  constants
    $reqdocs_i \in \mathcal{P}$(DOC)
    $mydoc_i \in$ DOC
  react $reset()$
    skip
  react $accept(d \in \mathcal{P}$(DOC) $\wedge\, l \in$ OFFICE$)$
    when $myloc_i = l \wedge reqdocs_i \subseteq d$
      say $save(mydoc_i)$
  react $notenough(d \in \mathcal{P}$(DOC) $\wedge\, l \in$ OFFICE$)$
    when $myloc_i = l \wedge reqdocs_i \not\subseteq d$
      say $goaway(deploc[reqdocs_i \setminus d])$
  react $dopers(d \in \mathcal{P}$(DOC) $\wedge\, l \in$ OFFICE$)$
    when $myloc = l$
      say do_pers
  react $alldone(d \in \mathcal{P}$(DOC) $\wedge\, l \in$ OFFICE$)$
    when $myloc_i = l \wedge reqdocs_i \subseteq d \wedge mydoc_i = finreg$
      say $leave\_vc$

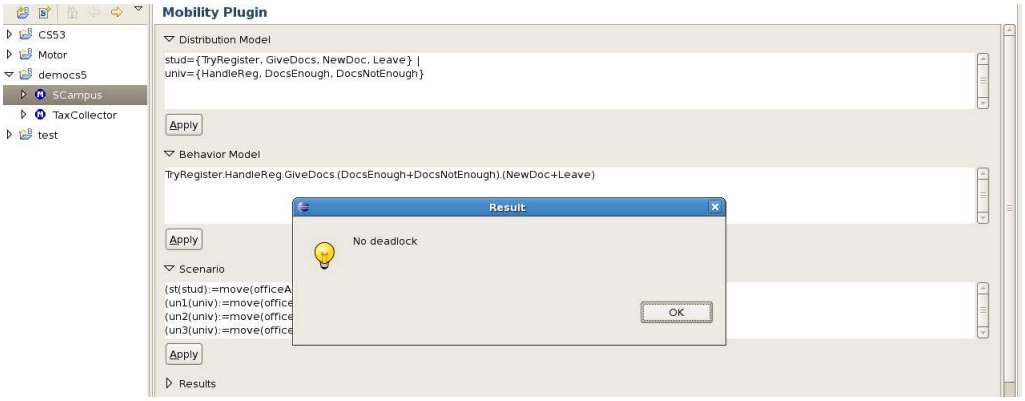
Code snippet from implementation of the student agent role in Java language with CAMA middleware using the reactive communication style. This code was written manually following the specification above.

```
class role_sa extends RoleSkeleton
    private VCOffice loc;
    private Vector<VCDoc> docs;
    private Vector<VCOffice> trace;

    public role_sa() {
        docs = new Vector<VCDoc>;
        race = new Vector<VCOffice>;
    }

    public void reactionMigrate(VCoffice l) {
        loc = l;
        post("Reset");
        actionAsk();
    }

    private void actionAsk() {
        post("Reply", new Record().add(docs).add(loc));
    }

    public void reactionSave(VCDoc newdoc) {
        if (trace.size() == 0) {
            l = random_new_loc();
            docs.add(newdoc);
       } else if (trace.size() != 0) {
            l = trace.last();
            trace.remove(l);
            post("Migrate", new Record().add(l));
       }
   }
   ...
}
```

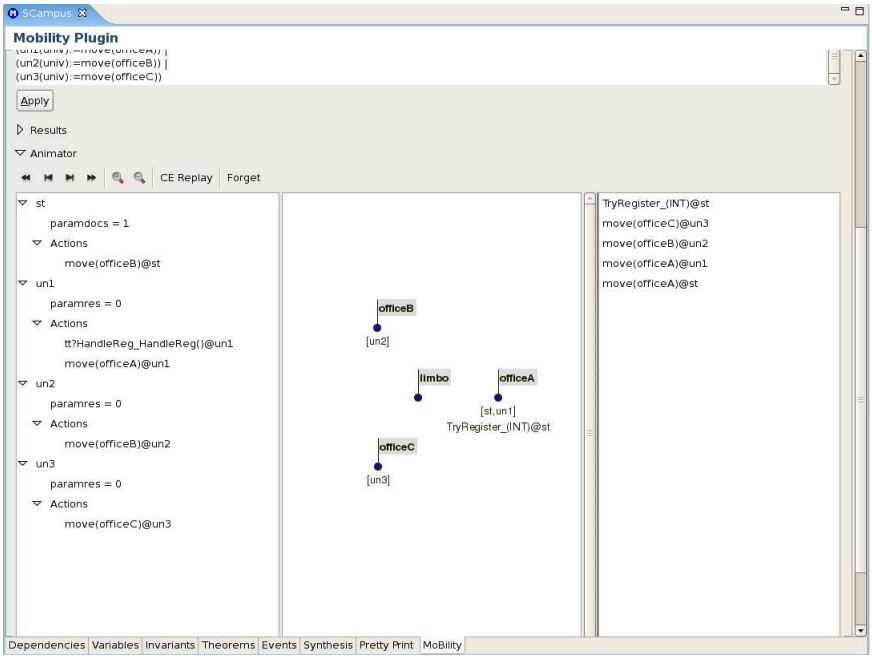
6.5 Screenshots of the platform and plugin use

The Mobility plugin is used by CS5 to verify system-wide dynamic properties. Sample screenshots show the Mobility plugin dialog pane within the platform with a loaded model. The Event-B part of the model is imported from the case study development while the behavioral and scenario parts are specific to the plugin.

On the first screenshot the tool reports absense of a deadlock in the checked model



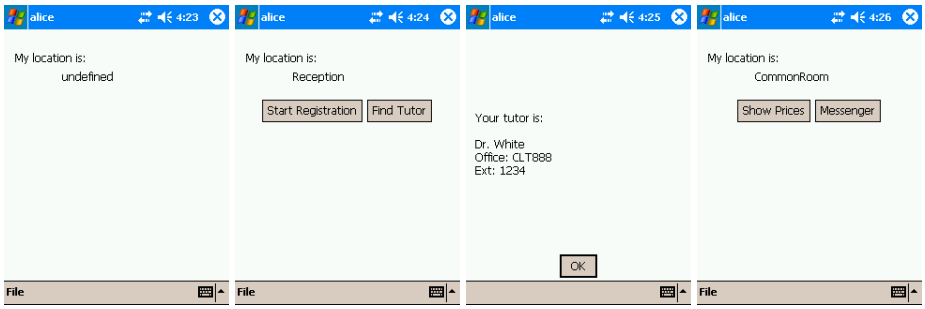
The second screenshot shows the same model analysed in the built-in animator. The animator interprets low-level model representaion which already combines Event-B specification behavioral model and scenario. The animator can also load a counter example (invariant violation or deadlock) that can be played to back to find the stem of a problem.



6.6 Demo

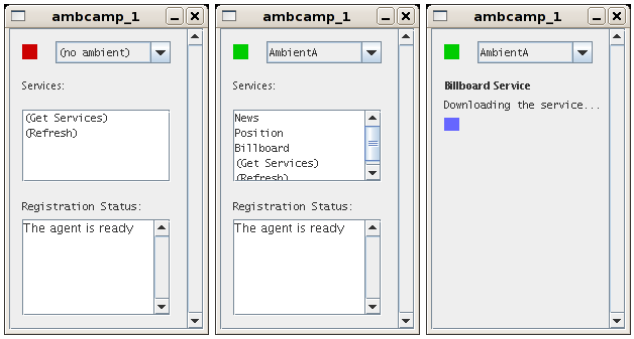
To implement ambients, we are incorporating smart dust motes into the case study scenario. These motes communicate with each other using Zigbee radio, and by customising the transmit power of the radio, we can use these motes as a localisation sensor. This enables us to deliver location-specific information and services to the users.

Each user carries a smart dust mote (with a unique id, so the mote acts as a badge, sort of speak) and a PDA as an interaction device. Each room is equipped with a smart dust base-station (receiver), which is connected to a controller application. The latter uses the CAMA middleware to communicate with the PDAs through Wi-Fi. When a user enters a particular room, his/her PDA shows the relevant information and/or services available for that room.



A set of rooms can be prepared to be smart dust aware. This can include a library, a meeting room, and an office. When a user enters a library, a list of newly acquired books can be displayed on the PDA, and the user can put his/her name to the waiting list through the PDA.

The part of the application concerned with automated registration does not have any graphical interface except the ability to announce results. The following screenshots from the main registration application shows typical stages of interaction between a user (student), student agent and ambients



6.7 Conclusion

Developing the case study we have applied a range of modelling and software engineering techniques. To design inter-agent interaction protocols we used CSP scenraious which were transformed into Event-B specifications with some syntax sugar. At the same time, we applied the mobility plugin to design and verify agent control logic - the part of an agent which glues one or more role implementations. The animation feature of the Mobility plugin helped to analyse behaviour prior to producing any executable code. The Pro-B animator integrated with platform was very usefull to understand role specifications. The CSP-based development of abstract role models helped to design agent communication protocols quicker than it can be done Event-B. However, we did transfer into Event-B development to make the full use of state-based modelling.

Bibliography

[1] Roman, G.C., Julien, C., Payton, J.: A Formal Treatment of Context-Awareness. In Wermelinger, M., Margaria, T., eds.: Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, LNCS 2984. Springer (2004) 12–36

[2] Marzo, G.D., Romanovsky, A.: Designing Fault-Tolerant Mobile Systems. In Guelfi, N., Astesiano, E., Reggio, G., eds.: Scientific Engineering for Distributed Java Applications International Workshop, FIDJI 2002, Luxembourg, LNCS 2604. Springer (2003) 185–201

[3] Iliasov, A., Romanovsky, A.: Structured Coordination Spaces for Fault Tolerant Mobile Agents. In Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A., eds.: LNCS 4119. (2006) 181–199

[4] Iliasov, A., Laibinis, L., Romanovsky, A., Troubitsyna, E.: Towards Formal Development of Mobile Location-based Systems, Presented at REFT 2005 Workshop on Rigorous Engineering of Fault-Tolerant Systems, Newcastle Upon Tyne, UK (http://rodin.cs.ncl.ac.uk/events.htm) (June 2005)

[5] C. Metayer, J.-R. Abrial, L.V., ed.: Rodin Deliverable D7: Event B language. Project IST-511599, School of Computing Science, University of Newcastle (2005)

[6] Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)