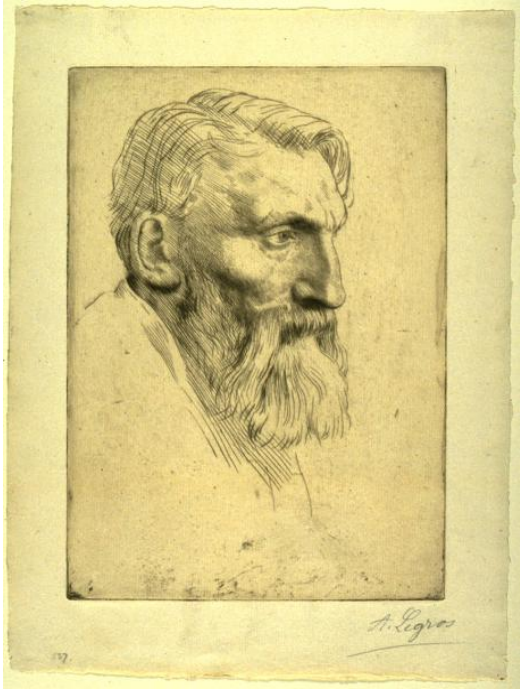
Project IST-511599
RODIN
"Rigorous Open Development Environment for Complex Systems"



RODIN Deliverable D11

Definition of Plug-in Tools

Editor: Michael Butler, University of Southampton

Public Document

31 August 2005

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

Contributors:

Michael Butler (University of Southampton)
Stefan Hallerstede (ETH Zurich)
Victor Khomenko (University of Newcastle)
Maciej Koutny (University of Newcastle)
Thierry Lecomte (ClearSy)
Sari Leppänen (Nokia)
Michael Leuschel (Heinrich-Heine-Universität Düsseldorf)
Vesa Luukkala (Nokia)
Manoranjan Satpathy (University of Reading)
Colin Snook (University of Southampton)

Table of Contents

1 Introduction

The RODIN open tools platform being developed in Workpackage 3 (WP3) will allow other parties to integrate their tools, such as model checkers and theorem provers, as plug-ins to support RODIN methods. This is likely to have a significant impact on future research in formal methods tools and will encourage greater industrial uptake of these tools. WP2 of RODIN is developing a collection of plug-in tools to be integrated in the RODIN platform. Developing these plug-in tools has two major aims:

- To provide extra functionality on top of the core platform to support more fully the application of the RODIN methodology being developed in WP2
- To validate the open architecture of the platform by populating it with a collection of plug-in tools covering a range of functionalities.

This deliverable (D11) describes our initial effort at defining a collection of plug-in tools. In the original proposal we identified the following plug-in tools for development within RODIN:

1. Linking UML and B
2. Petri net based model checking
3. Constraint-based model checking and animation
4. Model-based testing
5. Code Generation

Items 1-5 are covered in this report. In the original proposal we also allowed for further plug-ins to be identified and developed in the lifetime of the project. Three such additional plug-ins have been identified and are also covered in this report:

6. Graphical model animation
7. Documentation generation
8. Requirements manager

The identification and definition of plug-ins is driven by practical needs from the case study work. Identification of the graphical animation and documentation plug-ins was based on the experience of the partners, particularly ClearSy, on the application of B to large projects. The identification of the requirements manager was based on work in the Engine Failure Management case study.

These plug-ins are at varying levels of maturity. In some cases stand-alone prototypes of tools exist that provide some of the required functionality (1,2,3,5,6,7). An aim for these tools will be to produce versions of the tools that are properly integrated with the RODIN platform. In the other cases (4,8) the required functionality is being developed but specification and design of the tools has not commenced.

Early versions of the tools will be tested through application to the case studies of WP1, leading to improvements in functionality and design in the tools.

2 Linking UML and B

This section describes initial investigation of tools to provide a link between UML modelling and the B repository and tools. The link will consist of a specialisation of the UML defined by a UML profile called UML-B [SnookButler04a] and a tool, called U2B [SnookButler04b], that links UML-B models with the B database and tools being developed under Rodin. This link will be developed according to the following high level requirements and principles.

1) Functionality will be selectively based on the pre-existing U2B tool and UML-B profile.

2) A new version of the UML-B profile will be developed based on the UML2 metamodel plug-in project (http://www.eclipse.org/uml2/).

3) The UML-B profile will be developed to take advantage of UML 2.0 features wherever this seems appropriate.

4) The U2B translation tools will be developed as eclipse plug-ins

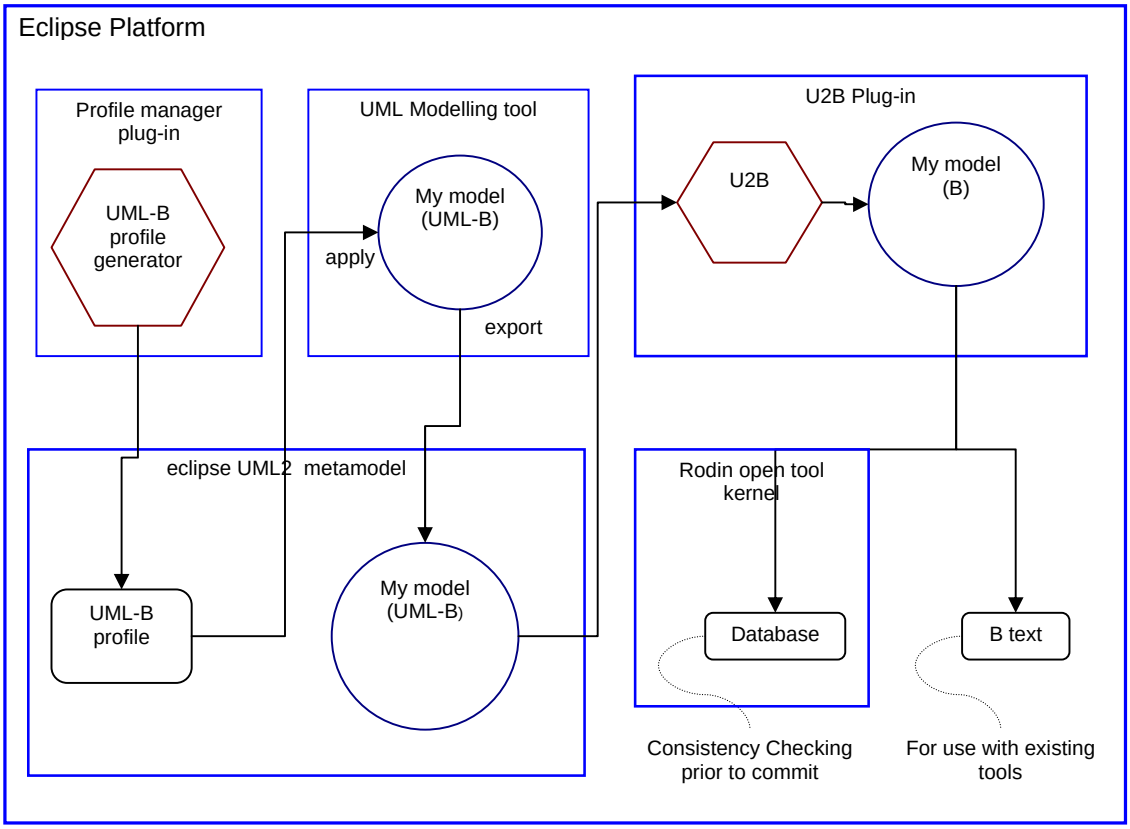
5) U2B will take as input, a UML model with the UML-B profile applied.

6) U2B will produce output by programmatically populating the B database.

7) As far as possible, UML-B and U2B will not be dependant on a particular proprietary modelling tool

Overview of organisation under eclipse

The proposed organisation of UML-B and U2B is illustrated in figure 1. All tools and resources are maintained within the eclipse environment. A profile manager plug-in will be developed which will programmatically generate a UML-B profile. An existing UML 2.0 modelling tool will be used to create UML-B models by applying the UML-B profile. The modelling tool will either be based on the UML2 metamodel (so that it uses UML2 as a repository) or, failing this, provide a facility to export to UML2 . The UML2 version of the model will then be converted into an XML version of B and stored in the Rodin B database.



UML-B

It is planned to use a UML profile to define the structure of UML-B. In UML 2.0 the profile extension mechanisms have been improved. Stereotypes now enable you to attach additional properties to entities. For example, we might wish to add an *invariant* property to some kinds of classes. A number of primitive types are available for properties (e.g. string, boolean, integer etc). Properties may also have a complex type that is defined by a class of the profile. A metamodel of UML 2 is available as a plugin project for eclipse (called UML2). The UML2 plug-in is based on EMF and provides a repository and management facilities, including a reflective editor, for manipulating models and profiles. It is intended to provide a basis for tool developers and therefore does not provide any modelling/visualisation facilities other than the reflective editor. The reflective editor is provided for initial experimentation only, it is intended that UML2 is used programmatically.

2.1 The UML-B profile

Mandatory stereotypes

Initially, we intended to use stereotypes to distinguish between the different roles of model elements. For example, <<*sees*>> and <<*refines*>> would be alternative stereotypes on a dependency. However, this requires the modeller to remember to apply a

stereotype on each element. Greater control, or at least indication, can be attained by forcing the application of a fixed stereotype for each and every modelling element and allowing the choices to be configured in the properties of the stereotype. Therefore, we structure the profile so that all (concrete) stereotypes are mandatory. This means that they are automatically applied to all instances of the model element that they extend. Properties are then used to distinguish different roles of UML entities within UML-B. For example <<*UBDependency*>> is a required stereotype that is automatically applied to all dependency relationships in the model. It has a property *kind*, which has an enumerated type: *dependencyET == sees*|*includes*|…|*refines*.

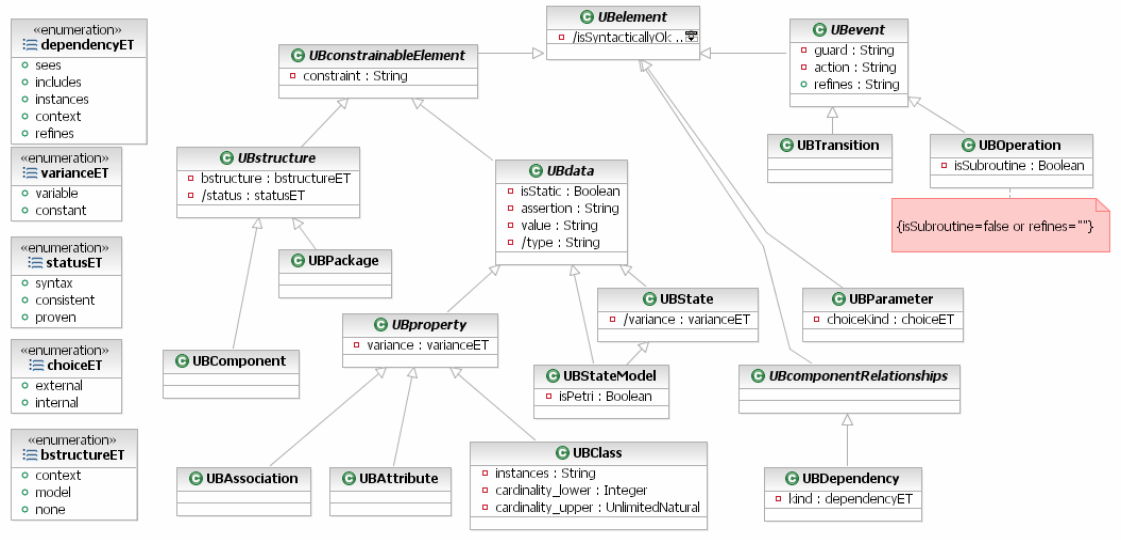
Inheritance of abstract stereotypes

There are many cases in UML-B where several different kinds of UML element require the same properties. For example, both class operations and state machine transitions require guards and actions. In order to avoid repetition of the definition of properties we used stereotype generalisation extensively in the profile. The generalised stereotypes are abstract (cannot be applied directly, only via their specialisations). In some cases several levels of generalisation of abstract stereotypes are defined. The resulting profile forms a tree structure of stereotypes with the leaves being the required stereotypes and the others being abstract.

Avoidance of UML features

In forming the stereotype properties that make up the profile, we decided not to rely on UML element properties, except for those diagram features that we define required stereotypes for. In some cases UML elements have standard text fields that have a similar intention to the properties we require. However, in previous work we have found that using these features leads to confusion because they may have slightly different semantics in UML leading the modeller to expect a different interpretation. Also, not all of these features are needed or appropriate in our modelling and many would be ignored. We have found that when some of the features are used, the modeller starts to assume that all the features have appropriate meanings in the translation. We therefore propose to replicate these features as properties in our stereotypes when necessary. This results in the simple rule that no UML model features are used except for the main elements that have a UML-B stereotype.

A drawing of the current profile stereotypes is given in Figure 2



2.2 Creation of Profile

Since maintaining profiles is difficult (once a profile has been applied to a model it can only be added to), we decide to maintain the profile via a programmatic profile creator. This is a java program that uses the UML2 API to create a profile. This will allow us to easily re-create the profile for installation, recovery etc. We could make copies of the program for each released version of the profile, so that old profiles can be re-created. It is easier to read a program that creates a profile than to read the profile itself. A plug-in has been deployed that creates the UML-B profile. The plugin works within RSA (Rational Software Architect) eclipse (the UML modelling tool that we are currently using) as well as the standard eclipse installation.
An alternative strategy that we investigated was to create and maintain the profile using the RSA profile editor. However we found that the RSA profile editor doesn't provide a graphical editor. Although it is a little better than the standard UML2 reflective editor we decided that it didn't give much benefit and opted for the profile manager plug-in

2.3 Applying profiles

Unfortunately, RSA does not use the UML2 metamodel plugin but has its own repository notation. RSA does have an export to UML2 and allows UML2 format profiles to be applied to its models. The fact that RSA doesn't use the UML2 repository means that we will not be able to perform real-time checking, parsing etc. It means that UML-B will still not be totally integrated with the B tools. For this reason, although we use RSA for the time being, we will continue to examine other alternatives as they arise. The UML-B profile must be generated outside of the RSA workspace so that when it is applied in RSA it is considered by RSA to be an external profile. In this case, when the model with applied profile is exported to UML2, the profile application is retained. Otherwise it will be lost.

2.4 Wellformedness and Parsing

A number of wellformedness constraints, are needed in addition to the profile information. For example we will require a constraint that the supplier and client of the dependency representing a *refines* relationship are both *UBstructure*s with the same *bstructure* property value. It is not desirable to define this constraint and apply it from the profile because it must be possible for the UML-B model to be inconsistent while it is being developed. We adopt the same philosophy as the B database: at creation, we only enforce the type of elements used, not their interrelationships. A secondary stage (initiated by a user action) will run a validation program to check wellformedness. (This might be performed when the U2B tool is about to be run or when the model is saved).

2.5 Structural features of UML-B

The existing version (U2B3) provides two mechanisms (packages and classes) for representing a B model (A B model is a machine, refinement or implementation). The class-model translation has been used less and less and could be dropped. However, the new version will be based upon UML2, which introduces 'components'. It may be useful therefore to switch to a component-model translation. In the first instance the package mechanism will continue to be supported for backwards compatibility.
(It is not clear how hierarchical classes (also introduced in UML2) relate to components. Some sources have claimed the two to be equivalent but we assume that components have no concept of instances.)
The new B will be organised into two kinds of constructs: models and contexts. Contexts define and refine the constant data features (SETS, CONSTANTS and their PROPERTIES). Models define and refine the variable data elements and the events that alter them. In general, UML-B models will contain elements that generate sets, constants, variables, events within a single model. Therefore, U2B will sort a UML-B model into B models and B contexts. However, the same reasons for having a separate context could apply in the UML-B representation. For the time being we will retain the ability to designate a UML-B structure as a context. The UML-B profile will provide a bstructure property on the <<UBstructure>> stereotype for indicating whether the structural feature is a model, context or neither.

2.6 B modelling styles

U2B3 provides 3 styles of modelling corresponding to different styles of using B: normal, action and event. The new UML-B will support the new B only. The new B is event-based but will also incorporate facilities for parameterisation of events which is the main facility required in the action style B. (Arrays of events where the parameter is the array 'index' representing external choice).

2.7 Instance modelling options

U2B3 supports three main options for modelling class instances. All 3 should continue to be supported.

- Variable – this is the traditional O-O style where objects can be created and deleted dynamically.

- Fixed – found to be useful for embedded systems where classes often represent a small number of objects that permanently exist.

- Class utility – no instances

All options have been found to be useful and will therefore be supported in the new version. There is a special case (Singular) of 'fixed' where only one instance exists. This has been less useful – it was introduced to try to improve ease of proof – but it is not clear that this is the case. We will decide whether to continue to support this or not.

U2B

The new U2B translator will be an eclipse plug-in. It will be structured upon a model of Event B. Each class in the Event B model will have a constructor that accepts as input parameter, the UML-B model (or relevant parts thereof) to be translated. The hierarchical structure of the model will result in a cascaded invocation of constructors starting from the top level constructor that creates an entire B project based on a complete UML-B model. The model will also be used to generate the database. It may be desirable to maintain separation of the database generation from U2B. This will be done by extending the basic database class with a class that adds the U2B constructor.

Summary of plug-in resources:

- ac.soton.umlb.profile_generator - The **profile manager** plugin tool
- ac.soton.umlb.u2b - The **U2B** translator tool
- ac.soton.profiles.umlb - the **UML_B** profile plug-in

2.8 UML Model Transformation in Lyra

The Lyra method developed at Nokia Research Center [LeppänenEtAl04] supports model-based approach in the development of distributed communicating systems and communication protocols. Although the method covers all stages of systems development, it is used particularly in the development of system architecture descriptions and implementation specifications. It supports service-oriented approach and the main technique used in the method is model decomposition/composition. The method consists of the four phases: Service Specification, Service Decomposition, Service Distribution and Service Implementation. An important objective is to integrate formal methods into the existing development process in Nokia by providing automatic translation of UML2-based Lyra design flow into the formal framework. A degree of automation in translating UML2 models and model transformation is perceived by Nokia as major criteria for evaluating the success of RODIN.

The high-level and implementation independent Service Specification model specifies the structure, interfaces and behavior of the services provided by the system. In Service Decomposition phase these are refined top-down and in step-wise manner into a set of functional specifications for the system level services. Service Decomposition models reflect the chosen implementation architectures but they are completely independent of

the implementation techniques to be used, and also of the chosen, underlying platform architecture. In Service Distribution phase the system level services, or their parts, are distributed over a given platform architecture, e.g. network architecture consisting of network elements. Distribution is part of the system implementation and thus the distribution should be transparent to the users of the system level services. To preserve the user's view of system level services as specified in the earlier phases, *communication protocols* are defined to implement the peer-to-peer communication. Peer-to-peer communication is defined between the distributed service parts as virtual communication, i.e. as independent of the chosen, underlying communication medias. In Service Implementation phase the distributed services are adapted and realized in their target environments (dynamic process management, process interaction etc.) and the virtual communication is realized using the communication medias and mechanisms provided by the underlying platform (e.g. routing, data encoding and decoding, adaptation to data transmission services provided by the underlying protocol layers).

The Lyra approach is based on producing a sequence of gradually refined specifications. To provide a sound and consistent development framework the design approach has to be accompanied by rigorous verification and testing methods. The Service Specification model(s) providing the correctness criteria for the later development phases can be verified using model-checking techniques. In the following development phases, currently algorithmic verification is used to verify the correctness of the decomposition and refinement steps. In many cases the Lyra design method is used only partially (e.g. ignoring the Service Implementation phase) for producing architecture specifications for (a set of) system implementations. Especially in these cases verification and testing of conformance between the architecture specification and implementation components is important.

Since the models describe the complete specification for services and service parts, abstraction of data and behavior according to the properties to be verified or tested is necessary.

The industrial-strength system case study will be centered on development of a *Position Calculation Application Part (PCAP)* specified by the *Third Generation Partnership Project (3GPP)*. PCAP is part of the *User Equipment (UE)* positioning system in a *UMTS (Universal Mobile Telecommunication System)* radio access network. PCAP is specified to manage the communication related to positioning service between the network elements *Radio Network Controller (RNC)* and *Stand-alone Assisted Global Positioning System Serving Mobile Location Center (*SAS). The case study represents a typical example of systems developed in Nokia and hence constitutes a valid test-bench for methods and tools developed within RODIN.

The specifications used in verification and testing are the models or their applicable parts, all developed using the Lyra method. Though the Lyra method is language independent, the main development language used in Nokia is UML2 language. Lyra defines a subset of UML2, which is called as *Lyra/UML2 profile*. The profile describes the concepts used in the systems development and the relationships between these concepts.

Lyra/UML2 profile has been developed in Nokia Research Center. It can be used as an input in the development of language profiles in Rodin. In WP1/CS1 the UML2 language is considered as the main modeling language, i.e. the main source language for verification and testing. The target languages, i.e. the languages used in verification and testing should be compatible with the Rodin tool platfom. Automated model transformation tool(s) for UML2 models into the (set of) chosen verification and testing language(s) should be developed in Rodin.

3 Petri net based model checking

The complexity of verification of concurrent and distributed systems is widely recognised as a major stumbling block in this key area of computer system design. One way of coping with the complexity problem is to use formal methods supported by computer aided verification tools. Within this approach, a well-established method is model checking [Clarke'99] which is completely automatic and relatively fast compared to other alternatives. It is therefore particularly attractive in industrial context as it can contribute successfully to the reduction of product development costs [Pixley'04].

Model checking is a technique in which the verification of a system is carried out using a finite representation of its state space. Basic properties, such as absence of a deadlock or satisfaction of a state invariant (e.g. mutual exclusion), can be verified by checking individual states. More subtle properties, such as guarantee of progress, require checking for specific sequences of states. Properties to be checked are typically described by formulae of a branching time or linear time temporal logic. An important pragmatic feature of model checking algorithms is that they produce counterexamples which can be used for debugging [Pixley'04].

Industrial strength model checkers are beginning to have an impact on practical designs and design methodologies. For example, in a "classical" reactive system application to the call processing software of a telephone switch at Bell Labs, model extraction combined with model checking revealed ten times as many concurrency related defects in the target code as the conventional system testing did [Holzmann'00]. Such an approach is particularly effective in detecting inter-process communication problems at an early stage of system design, helping to resolve the issue of design productivity.

Model checking of concurrent systems is intrinsically hard, and exhibits a trade-off between the compactness of the representation of the system and resources it takes to verify behavioural properties. For example, the classical deadlock detection problem is PSPACE-complete for a compact (bounded) Petri net or equivalent process algebra representation, but polynomial for transition system representation. However, the latter is often exponentially larger, and soon becomes too large to be stored in the main memory, which makes the algorithm impractical.

3.1 The problem of model checking mobile computing systems

Mobile systems are highly concurrent causing a state space explosion when applying model checking techniques. One should therefore use an approach which copes well with such a problem, in our case, based on partial order semantics of concurrency and the corresponding Petri net unfoldings [McMillan'92]. A finite and complete unfolding prefix of a Petri net *PN* is a finite acyclic net which implicitly represents all the reachable states of *PN* together with transitions enabled at those states. Efficient algorithms exist for building such prefixes [Khomenko'03], and complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent systems, because they represent concurrency directly rather than by multidimensional "diamonds" as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will be isomorphic to the net itself. Since mobile systems usually exhibit a lot of concurrency, their unfolding prefixes are often much more compact than the corresponding state graphs.

3.2 Model checking mobile systems using Petri net unfoldings

There exist several programming notations and frameworks proposed in the past to formally model and analyse the locality and movement of components, such as data and code. They are often based on, or directly inspired by, the π-calculus [Milner'92], which in itself is an extension of the CCS process algebra used in a variety of situations where reasoning about the behaviour of distributed communicating components is needed. As a result, π-calculus plays a foundational role in the continuing development of theories and methods for mobile and dynamically reconfigurable computing systems. Within RODIN, the plan is to develop a model checker based on Petri net unfoldings aimed at verifying π-calculus specifications, with the following main architectural components of the target plug-in:

- *translator* from π-calculus expressions used in the modelling of mobile systems to Petri nets
- formula *editor* where the user specifies the property to be verified
- *unfolder* for deriving a finite prefix of the unfolding of the translated Petri net
- *verifier* which establishes, by working on the finite prefix, whether the formula property is true of the original π-calculus input.

The model-checking plug-in for mobility systems is being developed and evaluated in the context of the mobile telecoms systems and the Ambient Campus case studies.

3.3 Mobility Plug-in and Ambient Campus

Mobile agent systems are increasingly attracting attention of software engineers. However, issues related to fault tolerance and exception handling for such systems have not received yet the level of attention they deserve. In particular, formal support for validating the correctness and robustness of fault tolerance properties is still under-developed. To address this issue, the work on the Petri net based model checking was planned to be conducted in close cooperation with the RODIN Ambient Campus case study.

Ambient Campus uses the CAMA system (context-aware mobile agents [Iliasov'05a] which is still under intensive development) that consists of a set of *locations*, and active entities of the system, called *agents*. An agent is a piece of software which is executed on a *platform*, providing execution environment interface to the location middleware. Agents can only communicate with other agents in the same location. Agents can migrate logically (connection and disconnection) or physically (e.g., movement of a PDA on which the agent is hosted) from a location to a location. Agents can also migrate logically from platform to platform using weak code mobility (transfer of application code or its parts from one host to another without retaining the execution state). Compatible agents (i.e., agents capable of cooperation in certain conditions in order to achieve individual agent goals and in accordance to the abstract specification of the whole system) collaborate through a scoping mechanism. Scopes define joint activities of several agents. Scoping mechanism also isolates non-compatible agents from each other.

As a result of adopting the CAMA system as a primary provider of model checking instances, and due to the technical decisions made in the design and implementation of latter, the scope of the Petri net based model checking has been extended to cover features related to asynchronous message passing (in addition to the synchronous message passing supported by the π-calculus). In concrete terms, it has been decided to add a capability of model checking designs expressed in a π-calculus based process algebra supporting constructs coming from the KLAIM system [Bettini'03]. The current details of the exact formalisation of the syntax and semantics of this extension are reported in [Iliasov'05]. It should be stressed, however, that the main architectural components of the target plug-in will remain the same, but they will be based on an input language which is closer to the intended application domain, in particular, to the development work within the Ambient Campus case study.

In concrete terms, our approach is first to give a formal semantics (including a compositional translation) of a suitably expressive subset of CAMA in terms of an appropriate process algebra and its associated operational semantics. The reason why we chose process algebra semantics is twofold: (i) process algebras, due to their compositional and textual nature, are a formalism which is very close to the actual notations and languages used in real implementations; and (ii) there exists a significant body of research on the analysis and verification of process algebras. The next steps, translation to a suitable Petri net formalism and model checking of the resulting Petri nets, will be supported by the mobility plug-in and unfolding based verification toolkit. Some work is still needed in order to clarify the range of properties that the mobility plug-in will be designed to verify in an efficient way.

3.4 Specific technical progress already achieved within RODIN

The main features of the syntax and semantics of the programming notation derived from CAMA have now been identified [Iliasov'05]. As already mentioned, in terms of the underlying model, it is based on KLAIM and the π-calculus.

The theoretical and algorithmic foundations of the compositional translation from the π-calculus to Petri nets, first developed for its finite fragment [Devillers'04], have recently been extended to a full recursive variant of π-calculus [Devillers'05a]. The ongoing (and

nearing completion) work aims at extending the previous developments to the KLAIM based process algebra [Devillers'05b].

Further development of algorithms needed for efficient implementation of the model-checking kernel of the mobility plug-in has been proposed in [Khomenko'05a]. The paper introduces a new condensed representation of a Petri net's behaviour which copes well not only with concurrency, but also with other sources of state space explosion, such as sequences of non-deterministic choice. Moreover, this representation is sufficiently similar to the traditional unfoldings, so that a large body of results developed for the latter can be re-used. Experimental results indicate that the proposed representation of a Petri net's behaviour alleviates the state space explosion problem to a significant degree and is suitable for model checking.

For a system developer, a crucial part of any model checking approach is information about system traces leading to an error state. Moreover, in order to be useful for debugging, such a trace should be as short as possible. The paper [Khomenko'05b] describes a new efficient method for computing the shortest violation traces in the Petri net unfolding approach.

3.5 Key requirements for the mobility plug-in

- **Input:**
  System specification in the form of a term of a suitable process algebra capable of capturing locality of code and data, migration of code and data, dynamic creation of concurrently operating processes
- *Property specification* in the form of a formula of a suitable temporal logic capable of capturing properties relating to locality and mobility, as well as fault-tolerant issues identified by the Ambient Campus study. A crucial aspect here is to identify a set of properties which can be model checked in an efficient way.
- **Model_translation:**
  Automatic translation from process expression to Petri nets is a pre-requisite to the subsequent model checking. The translation should lead to a suitable class of high-level Petri nets, with features allowing a direct and unambiguous linking of Petri net components to the sub-expressions of the original process expression.
- **Unfolding:**
  Applying the unfolding (and truncating) algorithm to the result of model translation yielding a finite prefix comprising the necessary information used in property verification.
- **Verification:**
  Applying property verifier to the finite prefix and property specification provided as user input. If the result is negative (i.e., the property is not satisfied) debugging information should be generated in the form of a counterexample.
- **Counterexamples:**
  A counterexample invalidating a given property should be returned to the user in way allowing easy debugging, for example, as an input to a simulator or a

visualisation sub-module. Any information returned should refer to the original system specification in a direct way.

Additional requirements for the mobility plug-in

- **Translation:**
  Automatic translation from CAMA to the process algebra used in the plug-in together with suitable linking of the corresponding part of the code (to allow direct interpretation of debugging information).
- **Unfolding:**
  Applying the unravelling algorithm to the result of model translation leading to a merged process.

4 Constraint-based model checking and animation

ProB is an animation and model checking tool for the B method. ProB's animation facilities allow users to gain confidence in their specifications. ProB contains a temporal and a state-based model checker, both of which can be used to detect various errors in B specifications. ProB supports checking of specifications written in a combination of CSP and B. ProB also supports automated refinement checking. In this section we provide an overview of the current functionality of ProB. We then give an overview our expectations of how ProB will be further developed in the RODIN project both in terms of integration with the RODIN tool environment and enhancement of the functionality of ProB.

4.1 Automated consistency checking for B

B is based on the notion of abstract machine. The variables of an abstract machine are typed using set theoretic constructs such as sets, relations and functions. Each machine has a certain number of operations that can update the variables of the machine, as well as an invariant specified using predicate logic. There are two main proof activities in B: *consistency checking*, which is used to show that the operations of a machine preserve the invariant, and *refinement checking*, which is used to show that one machine is a valid refinement of another.

The ProB animator and model checker has been presented in [LeuschelButler03]. Based on Prolog, the ProB tool supports automated consistency checking of B machines via exhaustive state space exploration. For exhaustive model checking, the given sets must be restricted to small finite sets, and integer variables must be restricted to small numeric ranges. This allows the checking to traverse all the reachable states of the machine. ProB can also be used to explore the state space non-exhaustively and find potential problems. The user can set an upper bound on the number of states to be traversed or can interrupt the checking at any stage. ProB will generate and graphically display counter-examples when it discovers a violation of the invariant. ProB can also be used as an animator of a B specification. So, the model checking facilities are still useful for infinite state machines,

not as a verification tool, but as a sophisticated debugging and testing tool. ProB comprises various visualization facilities [LeuschelTurner05] to display the state space in a user-friendly way.

The interactive proof process with Atelier-B or the B-Toolkit can be quite time consuming. A typical development involves going through several levels of refinement to code generation *before* attempting any interactive proof [Lanet02]. This is to avoid the expense of reproving POs as the specification and refinements change in order to arrive at a satisfactory implementation. We see one of the main uses of ProB as a complement to interactive proof in that errors that result in counterexamples should be eliminated before attempting interactive proof. For finite state B machines it may be possible to use ProB for proving consistency without user intervention. We also believe that ProB can be very useful in teaching B, and making it accessible to new users. Finally, even for experienced B users, ProB may unveil problems in a specification that are not easily discovered by existing tools.

ProB provides two ways of discovering whether a machine violates its invariant:

- It can find a sequence of operations that, starting from a valid initial state of the machine, navigates the machine into a state in which the invariant is violated. Trying to find such a sequence of operations is the task of the ProB *temporal model checker*.
- It can find a state of the machine which satisfies the invariant, but from which we can apply a *single* operation to reach a state which violates the invariant. Finding such states is the task of the ProB *state-based model checker*.

4.2 Automatic Refinement Checking

Refinement is a key concept in the B-Method. It allows one to start from a high-level specification and then gradually refine it into an implementation, which can then be automatically translated into executable code. While there is tool support for proving refinement via semi-automatic proof (within Atelier-B [Atelierb96], the B-toolkit [Btoolkit99], and now also B4Free), there has been up to now no automatic refinement checker in the style of FDR [FDRManual] for CSP [Hoare85, Roscoe98]. Thus, especially the development of B refinements has been a labour intensive activity. Indeed, when a refinement does not hold it may take a while for a B user to realise that the proof obligations cannot be proven, resulting in a lot of wasted effort. We wish to speed up B development time by providing an automatic refinement checker that can be used to locate errors before any formal refinement proof is attempted. In some cases the refinement checker can actually be used as an alternative to the prover[1], but in general the method is complementary to the traditional B tools.

[LeuschelButler05] formalises the notion of trace refinement checking and presents an algorithm which is at the heart of an automatic refinement checker. This new refinement

[1] Namely when all sets and integer ranges are already finite to start with and do not have to be reduced to make animation by ProB feasible.

checker has been implemented and integrated within the ProB tool. To compute the set of reachable states of a B machine the model checker makes use of the same underlying interpreter as the animator. In fact, the ProB interpreter can be viewed as providing the operational semantics of a B machine. We will re-use the same ProB interpreter as the foundation of the refinement checker. In case refinement is violated, the refinement checker displays a sequence of operations that can be performed by the “refinement'” machine but not by the specification.

4.3 Combining B and CSP in ProB

In the Event B approach [RodinD7], a B machine is viewed as a reactive system that continually executes enabled operations in an interleaved fashion. This allows parallel activity to be easily modelled as an interleaving of operation executions. However, while B machines are good at modelling parallel activity, they can be less convenient at modelling sequential activity. Typically one has to introduce an abstract `program counter' to order the execution of actions. This can be a lot less transparent than the way in which one orders action execution in process algebras such as CSP [Hoare85]. CSP provides operators such as sequential composition, choice and parallel composition of processes, as well as synchronous communication between parallel processes.

Our motivation is to use CSP and B together in a complementary way. B can be used to specify abstract state and can be used to specify operations of a system in terms of their enabling conditions and effect on the abstract state. CSP can be used to give an overall specification of the coordination of operations. To marry the two approaches, we take the view that the execution of an operation in a B machine corresponds to an event in CSP terms. Semantically we view a B machine as a process that can engage in events in the same way that a CSP process can. The meaning of a combined CSP and B specification is the parallel composition of both specifications. The B machine and the CSP process must synchronise on common events, that is, an operation can only happen in the combined system when it is allowed both by the B and the CSP.

In [Leuschel01] we presented the CIA (CSP Interpreter and Animator) tool, a Prolog implementation of CSP. As both ProB and CIA are implemented in Prolog, we were provided with a unique opportunity to combine these two to form a tool that supports animation and model checking of specifications written in a combination of CSP and B. We envisage two main uses of the combined tool. Firstly it can be used to animate and model check specifications which are a combination of B and CSP. We illustrate this below. The second use of the tool is to analyse trace properties of a B machine. In this case the behaviour is fully specified in B, but we use CSP to specify some desirable or undesirable behaviour and use ProB to find traces of the B machine that exhibit that behaviour. More details on combining B and CSP may be found in [ButlerLeuschel05] including an comparison with related work on combining state based approaches such as B with process algebras such as CSP.

4.4 Case Studies

The existing ProB tool is already being used in the Engine Failure Management case study and the CDIS case study and is proving useful both for animation and consistency

checking. This is providing valuable feedback in understanding the way in which animation should be presented and driven and the sorts of properties that are useful to check. We expect it will be applied to other case studies as the tool matures.

4.5 Future Requirements

The animation and model checking functionality needs to be integrated to the RODIN kernel. The kernel should provide access to models and proof obligations. The ProB tool already uses XML as an intermediate format for B models so that this integration should be easily achievable using XML. If ProB was an integrated plug-in to the kernel, it should then be possible to apply the animation and automated checking features seamlessly as part of formal developments.

There is further enhanced functionality that we have already identified for the ProB animation and automated checking approach:

- Support for Event B especially refinements involving the introduction of new events
- Simultaneous animation of a model and its refinement
- Link with the U2B plug-in so that animation and automated checking can be applied to UML-B models
- Currently ProB works through exploration of the state space of models. A complementary approach would be to attempt to falsify unproved proof obligations. This could save much effort resulting from attempting to manually prove proof obligations that turn out to be unsatisfiable.
- Optimisation of exploration should be achievable through the application of symmetry reduction techniques. This is especially true at higher levels of abstraction where models are likely to contain many symmetries.

5 Model-based testing

Model based testing is an umbrella term for several possible techniques. The general idea is to define the tests using some description or modeling mechanism and let a program perform the testing based on that defined model. The aim is to produce testing with better coverage by letting the computer automatically generate more, longer and more complex tests from the model that clearly exceed coverage that can be achieved by manually written test cases. This can be thought of as automatic simulation of the model against the implementation. The effort for creating the model-based test suite should less than that required for manually written test cases of similar coverage. "Manually written test cases" mean here the current practice of writing a test case that typically aims to tests a single property and upon execution against the System Under Test (SUT) returns a verdict that indicates whether the SUT correctly implements that property. Execution of such a test case can be expressed as a trace. These test cases are typically written in programming languages like C, Java or TCL (using frameworks like JUnit for Java and Expect for TCL), but there are dedicated testing languages, like TTCN-3, as well.

The models should be used to describe what behaviour is supposed to be tested and leave the how to test part for a tester program to perform based on the model. This is rather useful as in complex cases, like for example testing of parallel systems where it is possible to specify the components of the system and let the tool compute the large number of possible interleavings. Another example is specifying a data range, which yields all integers (or, say, every fourth integer) between two integer values (0-255). Another possiblity is to yield only the min and max values of the given range. In terms of manually written test cases, a model repesents several test cases.

The model is specified in a modeling language, such as SDL, UML, Z or B using the concepts of those languages. SDL, for instance has concepts of parallel communicating state machines that consist of states and transitions that along with concepts that are similar to programming languages, like arithmetic operations and conditional expressions. Note, that it is possible to define such concepts using any programming language and then use those concepts within that language. This can be thought of as modeling as well and in large test suites such libraries may easily emerge as they ease the actual task of testing. However, just packaging and using common behaviour is not modeling as such. Also, implementation of concepts typically found in a modeling language is a large task.

Software testing is broadly classified into two categories: *structured* testing and *functional* testing [AdrionEtAl82, Beizer95]. Structured testing (or white-box testing) derives test cases from the structure of the implementation or part of the implementation. Such test cases are derived from a programmer's perspective with the aim of covering as much as possible the structure of the object under test. This approach is likely to miss out many bugs because it may give all the code coverage that we may need, but it may not give us all of the system coverage that users may expect. The test cases for functional testing (or black-box testing) on the other hand are written from a user's perspective. They are derived from the external specification of the software behaviour with no consideration given to the internal organisation, logic, control or data flow. Structured tests tell a developer that the code is doing things right while functional tests tell a developer that the code is doing the right things. Functional testing involves executing the implementation under test in relation to a set of test cases and examining the correctness of the generated output. In this context, we have the following issues:

- Generation of test cases: How to obtain test cases so that they cover all features of a requirement under all scenarios?
- Execution of the test cases: How to execute the test cases which are obtained from requirements or specifications? This may be a difficult task because even if the implementation preserved the intent of the requirement/ specification, it may not preserve the structure or the logic of the latter.
- Validation of test outcomes: Once we run the test cases, the program would produce some outputs. How to ensure that the results are correct?

5.1 Existing work

If the development process is formal, many of the above issues can be handled in a rigorous manner. Formal specifications precisely define the high level aspects of a

software while omitting the detailed structural information; they are more likely to encode all of the required functions and their scenarios. Therefore testers can use the underlying mathematical framework to generate, possibly mechanically, test cases for functional testing. Even if we obtain test cases from specifications, it may not be easy to use them to execute the implementation. This is because a high level functionality may have been implemented in a variety of ways, and the mapping between the high level functionality and the low level implementation may not be apparent to the tester. Consider an example of a test case being a sequence of high level operations at specification level, but this operation sequence may not map easily to the operations at the implementation level. Some authors have proposed the use of special mappings called representation mappings to bridge this semantic gap [RichardsonEtAl92]. In addition, there is the problem of non-determinism. The choice made by a non-deterministic operation may not correspond to the deterministic choice made by the implementation. And then how are we going to use a test case involving non-determinism?

When a system executes a test case, it produces an outcome, and the outcome is often interpreted by the tester to assign a verdict that the system has passed the test. This problem can be tackled by incorporating oracles into the testing process [RichardsonEtAl92, Weyuker92]. A test oracle determines if the system behaved correctly in relation to the test case. Test oracles are usually obtained from specifications. The outcome of a test case and the outcome obtained from a test oracle need to be matched to establish the equivalence between abstract outputs and concrete results. There are two issues in this context; first, there must be a mapping between the abstract state of the specification and the concrete state of the implementation, and second, there must be a mechanism to show their equivalence. The first problem can be solved by representation mapping; Antony and Hamlet [AntoyHamlet00] have discussed how the users could write explicit code for a representation mapping between the concrete data structures of C++ instance variables and the abstraction of the specification. And the second can be addressed though the use of probing or observation operations both at the abstract as well as at the concrete state levels.

Early work on specification based testing includes that of Hall [Hall91] in which he discussed partitioning the input space by examining predicates in the operations of Z specification [Spivey88]. The aim was to induce software correctness based on test results. The work by Dick and Faivre [DickFaivre93] is a major contribution to the use of formal methods in software testing in which they have discussed a strategy for generating test cases from model oriented formal specifications. A VDM [Jones90] specification has state variables and an invariant (*Inv*) to restrict the state variables. An operation, say *OP*, is specified by a pre-condition (*Pre*) and a post-condition (*Post*). The approach of Dick and Faivre is essentially to partition the input space of *OP* by converting the expression $Pre \wedge Post \wedge Inv$ into its Disjunctive Normal Form (DNF); and each disjunct of it represents an input subdomain of *OP*. Next, as many operation instances of *OP* are created as the number of non-contradictory disjuncts in the DNF. An attempt is then made to create a FSA (Finite State Automaton) in which each node represents a possible machine state and an edge represents an application of an operation instance. A set of test

cases are then generated by traversing the FSA where each test case is a sequence of operation instances.

The work of Dick and Faivre discusses only the mechanisation of the partitioning algorithm. Legeard et al. [LegeardEtAl02] have developed a tool called the BZ Testing Tool (BZTT) for deriving test cases from Z or B specifications. So far as B specifications are concerned, they assume (i) the specification consists of a single B machine, and (ii) all sets in the B machine are transformed into finite enumerated sets. The BZTT test case generator computes DNF forms of the operation precondition and postconditions pairs and uses the separate disjuncts to define partitions of the state and operation inputs. Boundary goals are also incorporated to generate further partitions. Given a boundary condition, Prolog search techniques are used to generate a test preamble. At a boundary state, all eligible operations are applied to generate test cases as sequences of operation instances. From the test cases, automatic test scripts are generated in the target language, and representation mappings are created manually. Because of problems due to non-determinism and those related to matching between abstract and concrete states, automatic verdict assignment was not implemented.

The work of Richardson et al. [RichardsonEtAl92] discusses the derivation and use of test oracles for checking test results in the context of multi-lingual and multiparadigm (formal) specifications. Test oracles are derived from specifications in conjunction with the derivation of test data in relation to some testing criteria. Test execution is monitored and the results are verified against oracles; sometimes the authors considered it useful to compare intermediate results in addition to the end results. To make verification possible, their approach constructs mappings between the name space of the implementation and the name space of the oracle (same as the name space of the specification). There are two kinds of mappings: control and data. Control mappings are between control points in the implementation and locations in the specification where the implementation and the specification should be in same state. Data mappings describe the transformation between the data structures in the implementation and objects in the specification. These mappings are also called representation mappings [LegeardEtAl02], and usually they are developed manually. The implementation state and the state changes are monitored at the pre-determined control points, and data mappings are used to establish the correspondence between the implementation and the specification state as oracle. The authors point out that many of the steps described could be automated.

[SatpathyEtAl05] describes the experimental ProTest tool, an automatic test environment for B specifications. ProTest is based on the ProB model checker for B. ProTest follows an approach similar to the one by Dick and Faivre [DickFaivre93] and generates test cases from B specifications by partition analysis of the state invariant and the operation preconditions of a specification. ProTest generates test cases by partitioning and exploring the state space. ProTest then simultaneously animates the specification and runs the implementation with respect to the test cases and assigns verdicts whether the implementation has passed the tests. ProTest has an interface for running Java Programs with respect to test cases, and to explore the execution states through the use of probing operations. The whole process is automatic; however, at this stage the test environment

imposes some restrictions on operation arguments and results. Arguments are required to be basic types and specifications are required to be deterministic.

5.2 Model abstractions in Lyra

Abstraction of behavior

The Lyra design method [LeppänenEtAl04] relies heavily on the theories of process algebraic specification and thus the notion of externally observable behavior has significant role in Lyra. Total behavior is categorized into externally observable behavior and internal behavior.

Externally observable behavior consists of PSAPCommunication, USAPCommunication and PEERCommunication, which are the different types of behavior related to communication with the environment. PSAPCommunication encapsulates the behavior related to communication between the system and its users. The behavior type USAPCommunication encapsulates the behavior related to communication between the system and external service providers. PEERCommunication is defined in distribution of the system level functionality and encapsulates the behavior related to communication between the distributed system parts.

Internal behavior consists of behavior types ExecutionControl and InternalComputation. ExecutionControl type of behavior manages the execution flow. InternalComputation encapsulates the algorithms and functionality related to internal computation of the system. These behavior types are separated to allow easy distribution and implementation independent, modular specification of higher-level behavior.

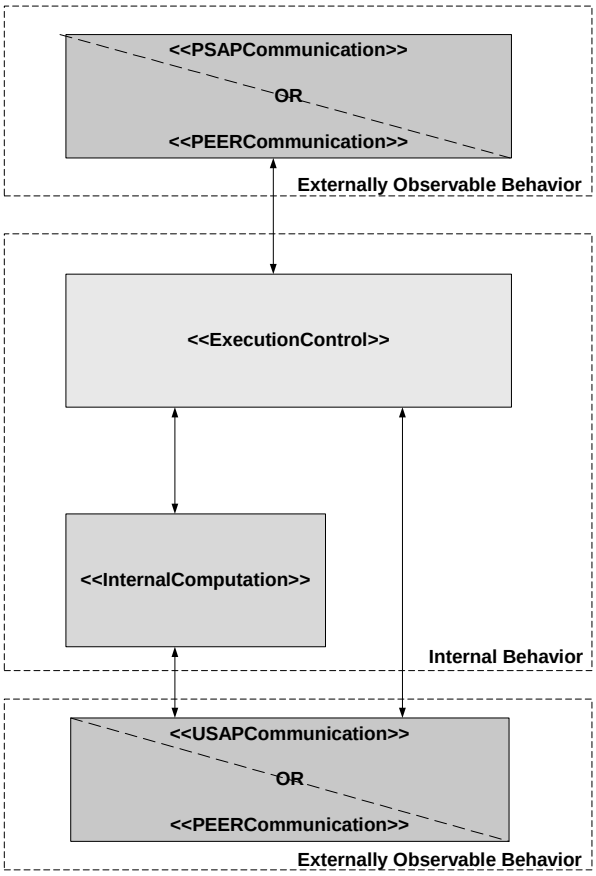


Figure 1: Lyra Behavioral Architecture (generic)

The different types of behavior have been specified in the Lyra profile. When using UML2 as the modeling language, stereotypes are used to label the model elements with the behavior types. Behavior is structured in the model using hierarchical state machines and classes for encapsulation. Layering, which implies the the triggering rules for encapsulated behavioral components has been illustrated in Figure 1: Lyra Behavioral Architecture (generic). This approach provides support for automated generation of abstracted verification and testing models from large and complex UML2 models.

In verification and testing the appropriate viewpoint of the system behavior for observation is chosen with respect to the properties to be verified and tested. When testing the externally observable behavior, by choosing the approapriate interfaces and sets of messages on those interfaces the tests can be clearly focused with respect to the chosen set of properties. The properties are related to a set of functionalities, which have the corresponding implementation also as internal behavior. Typing of the internal behavior provides support for abstraction of irrelevant internal behavior. The behavior, which is not relevant with respect to the set of specified test cases should be abstracted to obtain test models of appropriate size and complexity. Definition of the abstraction algorithms involves choosing the semantics approapriate for the test purpose (we assume the reduction algorithms to be part of the abstraction). For example, in Nokia the notion of CFFD equivalence and preorder have been used in abstractionverification and testing

with the TVT (Tampere Verification Tool). Different kind of abstraction tools can be developed as separate tools or embedded into model transformation tools.

Abstraction of data

One of the major challenges in testing and verification lies in the abstraction of data. Large or even infinite data domains make the systems hard or even impossible to verify and test them, at least exhaustively. In Rodin, different heuristics and patterns should be investigated and identified to make data abstractions possible and thus to manage the size and complexity of systems to be verified and tested.

Behavior related to communication is usually data independent: functionality in PSAPCommunication, USAPCommunication and PEERCommunication is related to receiving of incoming messages and sending of outgoing messages. Decisions and computation based on the information included in these messages is done in internal behavior, i.e. in ExecutionControl and InternalComputation.

In WP1/CS1 we have defined the system behavior through *abstract machines*, which correspond to the Lyra behavior types. *ACM (Abstract Communicating Machine)* handles communication, i.e. receiving and sending of messages on PSAP, USAP and PEER interfaces. This behavior type is usually data independent. Part of ACM corresponds to ExecutionControl, which identifies the message type and forwards the (appropriate part of the) message to be handled by the InternalComputation or (in case the computation algorithms resides in an external entitiy) to be set further by USAPCommunication or PEERCommunication. This behavior type manages the execution control flow and uses only the part of the messages indicating the message type. In UML2 also the *entry* and *exit points* with the related data parameters are used for managing the execution control flow. ACAM (Abstract CAlculation Machine) corresponds to InternalComputation behavior type and encapsulates the algorithms and other internal calculation. This behavior part is strongly data dependent. Outcome of the internal computation directs implicitly the execution (through the return values, i.e. exit points visible for ExecutionControl).

In the Lyra design method the ASN.1 kind of approach has been used to structure and encapsulate the data into message parameters. This approach supports abstraction of the irrelevant message parts from the incoming messages.

In managing the size and complexity of the system models to be tested, abstraction of data domains is crucial. In Rodin, heuristics and patterns for identifying suitable abstractions for large and infinite data domains should be investigated and developed. Also the possibility to abstract data variables from the test specification should be investigated (e.g. use of program slicing kind of techniques to define dependencies between the variables) . Abstraction of data in test specifications (produced using model transformation tools for UML2 models) should be computer-assisted or fully automated. To allow a degree of automation data abstraction tools for test specifications should be developed in Rodin.

5.3 Testing Plug-in

The results of testing are reported as test verdicts (e.g. pass, fail, inconclusive), preferably together with execution traces as counter-examples in failure (and possibly inconclusive) cases. Execution traces should be generated in such a form that can be interpreted with respect to the initial model, for example as sequences of messages. Possibly also the execution traces could be stored into a test log. If the testing tool is able to use the test logs as its own input, this would allow higher degree of automation when repeating the tests (e.g. regression testing). This would also allow the use of other, external model transformation and testing tools.

Test reports should also include information on the test coverage. Test coverage can be indicated e.g. with respect to states or transitions in the abstracted test specification. Reporting and generation of counter-examples should be developed and implemented as part of the testing tool and/or test execution environment.

Modeling of the environment should be considered. In verification and testing of communication protocols an abstracted model of the relevant part of lower layers behavior should be part of the test models. For example, an abstraction of the underlying media (e.g. radio interface) should be brought into the testing environment as part of the test system to allow automated test execution.

Another issue is how to simulate the inputs from the environment, for example messages sent by the upper layer(s). This is a typical way of triggering the behavior to be observed in the system under test. Triggering can also be done through user interaction. Solutions for these issues should be part of the model-based testing method and implemented in the testing tool.

For development of the testing algorithms the appropriate semantic models should be defined. Verification and testing of the specified properties is done with respect to the chosen semantics. Testing algorithms implement different kind of testing heuristics. The implementation of the testing tool and the execution environment should be modular and expandable. For example the testing algorithms could be stored into a library, which allows flexible use and easy adding of new algorithms.

We believe that the approach outlined above for generating test cases from formal specifications fits well into an overall package of tools to support the RODIN approach to system development. A plug-in to support model-based testing should try to meet the following requirements:

- Automate the generation and running of tests
- Provide flexible ways of defining and applying boundary cases
- Provide flexible and comprehensive ways of mapping the interface of a high level formal specification to the interface of an implementation under test.
- Deal with nondeterminism in specifications
- Support testing of a range of implementation languages. This is probably best achieved by using a standard test specification language such as TTCN

- Provide a metrics for the degree of test coverage and indications of the degree of coverage being achieved

These represent a very challenging set of requirements which will require a large research effort to achieve. It is unlikely that we will achieve a powerful general purpose model-based testing tool in the near future, but several of the pieces will be provided by RODIN to work towards such a tool. RODIN also has a range of challenging case studies which can be used to guide and validate any tool development for model based testing.

6 Code Generation

This plug-in aims at enabling the production of source code from a given Event B model. This production is performed in 2 steps:

- Events are recomposed by applying composition rules. Such rules are described in [Abrial01]. A rule is composed of:
  - an antecedent (the matching pattern)
  - a consequent (the resulting pattern)
  - a side condition which has to be verified by a theorem-prover before applying the rule.
- Final recomposed event is translated into target source code. Target source code could be C, C++ or Java.

The process flow is described in the following figure.

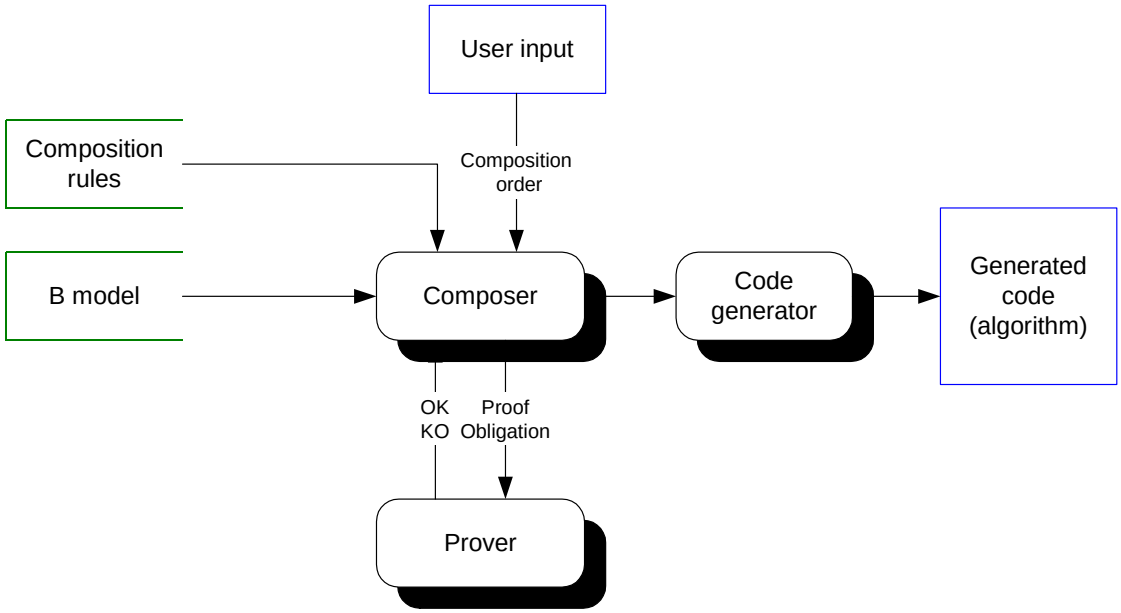


Figure 3 : code generation process flow

The Composer tool is an interactive tool, performing transformations ordered by a user. No automation is required. The tool will show up the composition rules that may be applied on some of the events.

Generated code would be either C, C++ or Java.
Required functions are:

- Load an event system
- Apply rules on events
- Search applicable rules
- Load and save recomposition rules
- Save the created system

6.1 Composition rules

A recomposition rule is applied on one or more events to replace them by one or no event. A recomposition rule has:

1. a name
2. a list of initial events
3. a created event
4. some proof obligations
5. some instantiable joker

The point 3,4 and 5 are optional: a rule can be used to delete some events, it can be used without proving anything and most of the time all the jokers are instantiated automatically. The order of the initial events is important: when trying to apply a rule on some events, we try to match them using this order.

6.2 Interface

4 viewers are expected:

- the list of events
- the list of rules or the list of possible rule applications
- the currently selected event
- the currently selected rule

With the menu and the tool bar, one can

- **Open a new file:** the file is loaded and the event list is updated.
- **Save to a file:** the current list of event is saved in a file with the prelude (variables, invariant, ...) as the initial one.
- **Change the preferences:** the list of preferences can be changed.
- **Load a rule file:** the file is loaded and the rule list is updated.
- **Save a rule file:** the current rule list is saved in a XML file
- **Apply a rule:** a rule is applied on selected events, the list of possible rule (with the same number of initial events as the number of selected events) is displayed in a window that allows also the change the order of the selected events in order to make them match the rule.

- **Search rule application:** the possible application of selected rules is searched on all possible combinations of events. The rule is displayed in the application viewer.
- **Edit the current events:** the current system file is edited using the editor specified in the preferences
- **Going back to the previous event list:** the previous event list before a rule application is restored.
- **Going forward to the next event list:** the next event list that have already been calculated is restored.

In the rule viewer, one can

- add, remove and modify some rules
- Search rule application by double-clicking on a rule

In the application viewer, one can apply a rule by double-clicking on an application.

7 Graphical model animation

Presenting a B model is never an easy task, as the receiver of this presentation is usually not able to read and/or understand a B model. This plug-in aims at providing presentation support to expose results of a formal modeling study, not limited to a text based representation (listing). The process flow is described in the following figure.

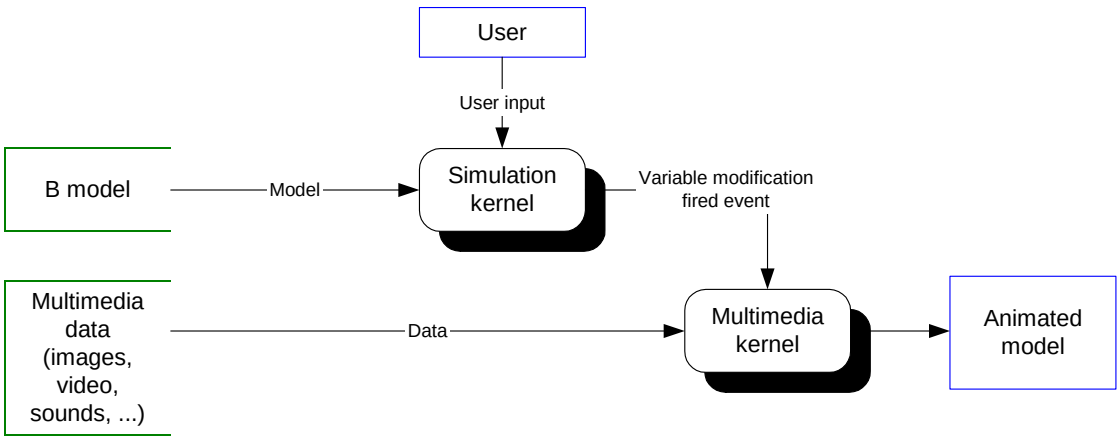


Figure 4: Model animation process flow

The very objective of this plug-in is to provide some attractive, easily understandable, meaningful, multimedia representation of a B model. This plug-in doesn't offer any debugging capabilities: the ProB model checker plug-in offers that service.

Required services:

1. **graphical presentation set up**: associate model modification with graphical animation.
2. **stepwise progress**: select an event to fire, provide value(s) to variable(s).
3. **starting point selection**: select any starting point (even unreachable).
4. **play a predefined scenario**: this scenario is described in a formatted text file (variables modified, events fired).

Software architecture

The animator is made of 2 components:

- a graphical interface: its role is to represent observations and to command the simulation kernel.
- a simulation kernel: this kernel is composed of 3 elements:
  - set of variables of the B model to animate,
  - events
  - a sequencer

  As a debugger, he has 2 main properties:
  - it can be commanded: initialized, stopped, launched in automatic mode, run in step by step mode and allowing the valuation of variables.
  - It can be observed: an external program may have access to any variable modification or to any enabled event.

8 Documentation generation

This plug-in aims at generating automatically some documentation related to a B model. The objective is to enable the reading of a formal model, but without any prior knowledge of B.

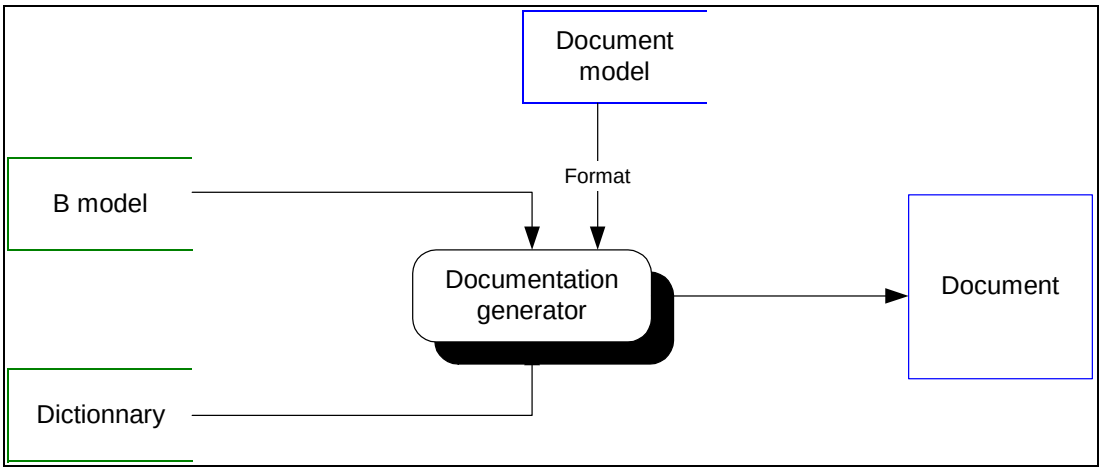


Figure 5: Documentation generation process flow

This kind of representation can be seen as a mid-term between multimedia presentation (see §**Error! Reference source not found. Error! Reference source not found.**) and the original, text-based B model.

Generating by program the very semantics of a model is impossible. So extra inputs are required from the user to automatically generate this documentation.

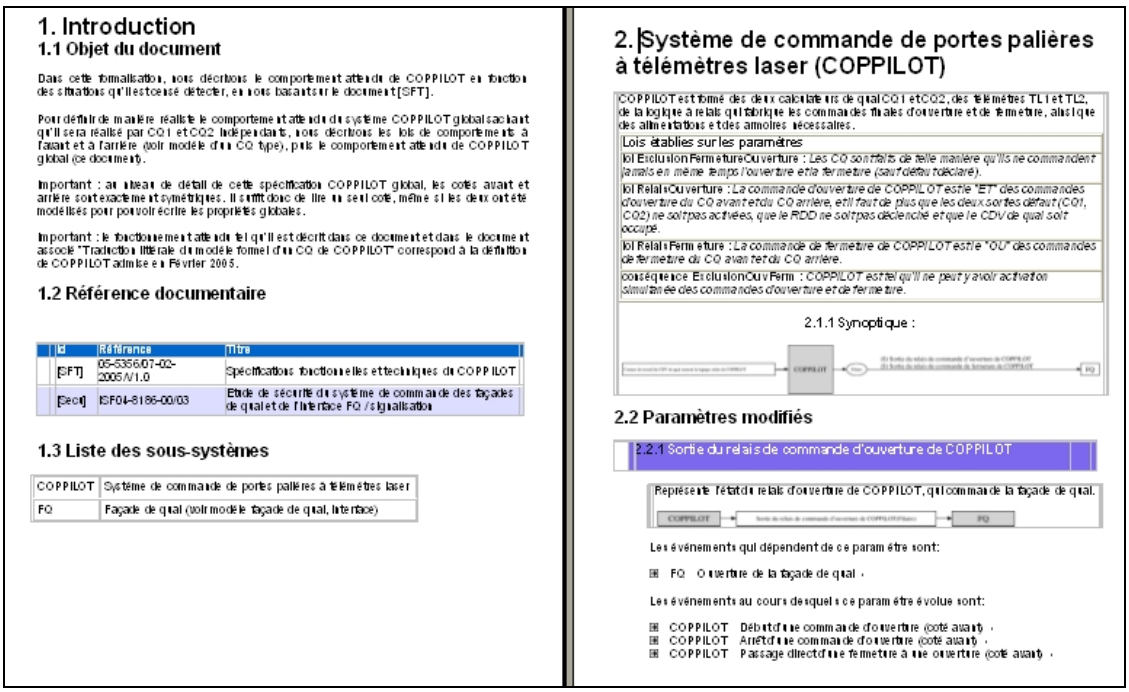
In fact, the B model should be released with a complementary document called "dictionary". Such a dictionary is composed of:

- The list of all components/subsystems composing the system being modelled,
- For each event:
  - The name of the component/subsystem which it is related to,
  - a brief description of this event,
  - a detailed description of it.
- a description of each variable and constant.

The resulting document should contain a description of the system being modelled, based on the relationships existing among all components/subsystems. For example, should be exhibited:

- impact of events on different subsystems,
- shared variables (variables modified/used by several subsystems),
- …

with graphical/text-based representation. An example is provided below, resulting from our previous experience.



1. Introduction

1.1 Objet du document

Dans cette formalisation, nous décrivons le comportement attendu de COPPILOT en fonction des situations qu'il est censé détecter, en nous basant sur le document [SFT].

Pour définir de manière réaliste le comportement attendu du système COPPILOT global sachant qu'il sera réalisé par CO1 et CO2 indépendants, nous décrivons les lois de comportements à l'avant et à l'arrière (voir modèle d'un CO type), puis le comportement attendu de COPPILOT global (ce document).

Important : au niveau de détail de cette spécification COPPILOT global, les cotés avant et arrière sont exactement symétriques. Il suffit donc de lire un seul coté, même si les deux ont été modélisés pour pouvoir écrire les propriétés globales.

Important : le fonctionnement attendu tel qu'il est décrit dans ce document et dans le document associé "Traduction littérale du modèle formel d'un CO de COPPILOT" correspond à la définition de COPPILOT admise en Février 2005.

1.2 Référence documentaire

Id	Référence	Titre
[SFT]	05-5356/07-02-2005/V1.0	Spécifications fonctionnelles et techniques du COPPILOT
[Secr]	ISF04-8186-00/03	Etude de sécurité du système de commande des façades de quai et de l'interface FQ /signalisation

1.3 Liste des sous-systèmes

COPPILOT	Système de commande de portes palières à télémètres laser
FQ	Façade de quai (voir modèle façade de quai, interface)

2. Système de commande de portes palières à télémètres laser (COPPILOT)

COPPILOT est formé des deux calculateurs de quai CO1 et CO2, des télémètres TL1 et TL2, de la logique à relais qui fabrique les commandes finales d'ouverture et de fermeture, ainsi que des alimentations et des armoires nécessaires.

Lois établies sur les paramètres

loi ExclusionFermetureOuverture : *Les CO sont faits de telle manière qu'ils ne commandent jamais en même temps l'ouverture et la fermeture (sauf défaut déclaré).*

loi RelaisOuverture : *La commande d'ouverture de COPPILOT est le "ET" des commandes d'ouverture du CO avant et du CO arrière, et il faut de plus que les deux sorties défaut (CO1, CO2) ne soit pas activées, que le PDD ne soit pas déclenché et que le CDV de quai soit occupé.*

loi RelaisFermeture : *La commande de fermeture de COPPILOT est le "OU" des commandes de fermeture du CO avant et du CO arrière.*

conséquence ExclusionOuvFerm : *COPPILOT est tel qu'il ne peut y avoir activation simultanée des commandes d'ouverture et de fermeture.*

2.1.1 Synoptique :

2.2 Paramètres modifiés

2.2.1 Sortie du relais de commande d'ouverture de COPPILOT

Représente l'état du relais d'ouverture de COPPILOT, qui commande la façade de quai.

Les événements qui dépendent de ce paramètre sont:

FQ Ouverture de la façade de quai

Les événements au cours desquels ce paramètre évolue sont:

COPPILOT Début d'une commande d'ouverture (coté avant)
COPPILOT Arrêt d'une commande d'ouverture (coté avant)
COPPILOT Passage direct d'une fermeture à une ouverture (coté avant)

Figure 5: example of generated documentation

Target format would be HTML, RTF and/or PDF.

9 Requirements Manager

A need for tool support for instantiating generic requirements patterns was conceived while working on case study 2. In this case study a generic model of a failure management subsystem was developed and then made specific to an example application by defining a set of instances. The specific example had been described previously in a tabular form which was then translated in terms of the generic model for verification. The instantiation was successful but tedious. Tool support for adding individual instances and maintaining the instantiation in the form of a database is envisaged. The tool will perform verification that the instantiation conforms to the constraints expressed in the generic model.
The tool will maintain a database representing the instance data for a line of software products. The product line is based on a generic requirements specification that is expressed as a UML-B class diagram. The following features are required:

1. A database schema for the instance data tables will be automatically generated from a class diagram generic model.

2. The tool will provide a context dependent menu extension to a UML-B modeling tool for adding class instances and corresponding table entries.

3. Database support (create/ update/ delete/ audit trail) will be provided at product and at instance level.

4. The requirements manager will provide a capability for batch data input as well as manual addition of individual instances and the capability to switch between the two.

5. The requirements manager will verify new/updated instance data against the generic model, including

    a. association multiplicities,

    b. constraints (written in μB, the action and constraint notation of UML-B) attached to UML-B entities as stereotype data fields.

6. Verification feedback notification will be given by annotating the violated constraints and the offending item in the table.

The tool will be implemented in Java and deployed as an Eclipse plug-in. The UML-B modeling tool to be extended will be Rational Software Architect (RSA).

10 References

[Abrial96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[Abrial01] J.-R. Abrial. *Event Driven Sequential Program Construction*. January 2001. www.atelierb.societe.com/documents_en.htm

[AdrionEtAl82] Adrion, W. R., M.A. Branstad and J.C. Cherniavsky, Validation, Verification and Testing of Computer Software, *ACM Computing Surveys*, Vol. 14(2), June 1982.

[Antoy Hamlet00] Antoy, S., and D. Hamlet, Automatically Checking an Implementation against its Formal Specification, *IEEE Transactions on Software Engineering*, Vol. 26(1), January 2000, pp.55–69.

[Atelierb96] Steria. *Atelier B, User and Reference Manuals.* Aix-en-Provence, France, http://www.atelierb.societe.com/index uk.html, 1996.

[Beizer95] Beizer, B., *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley, 1995.

[Bettini'03] L.Bettini et al.: *The KLAIM Project: Theory and Practice*. LNCS 2874 (2003)

[Btoolkit99] B-Core (UK) Limited. *B-toolkit manuals*. Oxon, UK, www.b-core.com, 1999.

[ButlerLeuschel05] Butler, M. and Leuschel, M. (2005) Combining CSP and B for Specification and Property Verification. In *Proceedings of Formal Methods 2005* (in press), Newcastle upon Tyne. Fitzgerald, J, Hayes, I. and Tarlecki, A., Eds. eprints.ecs.soton.ac.uk/10388

[Clarke'99] E.M.Clarke, O.Grumberg and D.Peled: *Model Checking*. MIT Press (1999)

[Devillers'04] R.Devillers, H.Klaudel and M.Koutny: *Petri Net Semantics of the Finite π-Calculus.* FORTE (2004, full version submitted 2005)

[Devillers'05a] R.Devillers, H.Klaudel and M.Koutny: *A Petri Translation of π-Calculus Terms.* CS-TR-887, University of Newcastle (2005)

[Devillers'05b] R.Devillers, H.Klaudel and M.Koutny: *A Petri Net Semantics of a Simple Process Algebra for Mobility.* Manuscript (2005)

[DickFaivre93] Dick, J., and A. Faivre, Automating the generation and sequencing of test cases from modelbased specifications, *Proc. of the FME'93: Industrial Strength Formal Methods Europe*, LNCS 670, 1993, pp. 268–284.

[FDRManual] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual.* www.fsel.com

[Hall91] Hall, P.A.V., Relationship between Specifications and Testing, *Information and Software Technology*, Jan/Feb 1991.

[Hoare85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[Holzmann'00] G.Holzmann and M.Smith: *Automating Software Feature Verification*.

Bell Labs Technical Journal (2000)

[Iliasov'05] A.Iliasov,V.Khomenko, M.Koutny and A.Romanovsky*: On Specification and Verification of Location-based Fault Tolerant Mobile Systems*. WREFT (2005)

[Iliasov'05a] A.Iliasov, L.Laibinis, A.Romanovsky and E.Troubitsyna: *Towards Formal Development of Mobile Location-Based Systems*. WREFT (2005)

[Jones90] Jones, C.B., *Systematic Software Development using VDM*, 2nd Edition, Prentice Hall, 1990.

[Khomenko'03] V.Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings.* PhD Thesis, University of Newcastle upon Tyne (2003)

[Khomenko'05a] V.Khomenko, A.Kondratyev, M.Koutny and V.Vogler: *Merged Processes - a New Condensed Representation of Petri Net Behaviour*. CONCUR (2005)

[Khomenko'05b] V.Khomenko: *Computing Shortest Violation Traces in Model Checking Based on Petri Net Unfoldings and SAT*. WREFT (2005)

[Lanet02] J-L Lanet. The use of B for Smart Card. In *Forum on Design Languages (FDL02)*, September 2002.

[LegeardEtAl02] Legeard, B., F. Peureux and M. Utting, Automated Boundary Testing from Z and B, *Proc. Of the FME'02 (Formal Methods Europe) Conference*, LNCS No. 2391, 2002, pp. 21–40.

[LeppänenEtAl04] Leppanen, S., Turunen, M. and Oliver, I., Application Driven Methodology for Development of Communicating Systems. *FDL'04, Forum on Specification and Design Languages*. Lille, France, September 2004.

[Leuschel01] M. Leuschel. Design and implementation of the high-level specification language CSP(LP) in Prolog. In I. V. Ramakrishnan, editor, *Proceedings of PADL'01*, LNCS 1990, pages 14–28. Springer-Verlag, March 2001. eprints.ecs.soton.ac.uk/7253

[LeuschelButler03] Leuschel, M. and Butler, M. (2003) ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003,* LNCS 2805, pages pp. 855-874, Pisa, Italy. Keijiro, A., Gnesi, S. and Dino, M., Eds. eprints.ecs.soton.ac.uk/8341

[LeuschelButler05] Leuschel, M. and Butler, M. (2005) *Automatic Refinement Checking for B*. Technical Report, School of Electronics and Computer Science, University of Southampton. eprints.ecs.soton.ac.uk/10907

[LeuschelTurner05] Leuschel, M. and Turner, E. (2005) Visualising Larger State Spaces in ProB. In *Proceedings of ZB 2005* LNCS 3455, pages pp. 6-23, Guildford, UK. Treharne, H., King, S., Henson, M. and Schneider, S., Eds. eprints.ecs.soton.ac.uk/10798

[McMillan'92] K.L.McMillan: *Symbolic Model Checking: an Approach to the State Explosion Problem.* PhD Thesis, Carnegie Mellon University (1992)

[Milner'92] R.Milner, J.Parrow and D.Walker: *A Calculus of Mobile Processes.* Information and Computation (1992)

[Pixley'04] C.Pixley: *Formal Verification 2004.* EDA Tools Forum (2004)

[RichardsonEtAl92] Richardson,D.J., S.L. Aha, T.O. O'Malley, Specification-based Test

oracles for Reactive Systems, *Proc. of the 14th ICSE,* ACM Press, 1992, 105–118.

[RodinD7] C. Métayer, J.-R. Abrial, and L. Voisin. *Event-B Language*. RODIN Deliverable 3.2, Project IST-511599, rodin.cs.ncl.ac.uk, 2005.

[Roscoe98] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice–Hall, 1998.

[SatpathyEtAl05] Satpathy, M., Leuschel, M. and Butler, M. (2005) ProTest: An Automatic Test Environment for B Specifications. *Electronic Notes in Theoretical Computer Science* 111:pp. 113-136.

[SnookButler04a] Snook, C. and Butler, M. (2004) *UML-B: Formal modelling and design aided by UML*. Technical Report, Electronics and Computer Science, University of Southampton. eprints.ecs.soton.ac.uk/10169/

[SnookButler04b] Snook, C. and Butler, M. (2004) *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.

[Spivey88] Spivey, J.M., *Understanding Z,* Cambridge University Press, 1988.

[Weyuker92] Weyuker, E.J., On Testing Nontestable Programs, The Computer Journal, Vol 25(4), 1982, pp. 465–470.