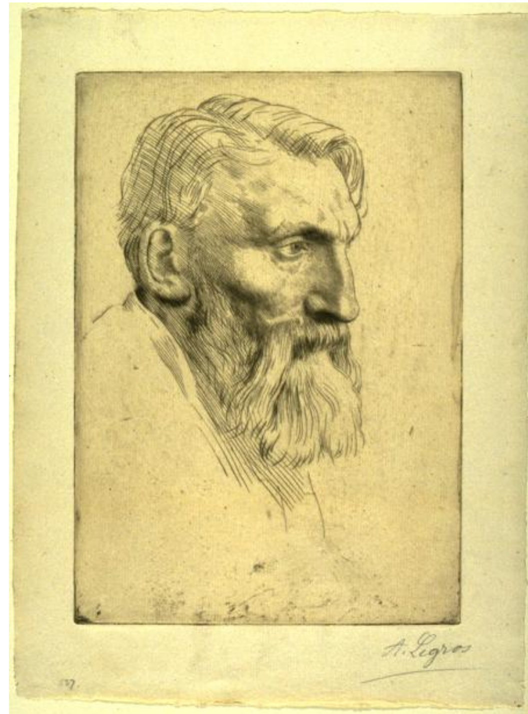
Information Society Technologies

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



RODIN Deliverable D9

Preliminary report on methodology

Editor: C. B. Jones University of Newcastle upon Tyne

Public Document

8th August 2005

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

Abstract

One aim of the Rodin project is to contribute *formal methods* which will underpin the creation of *fault-tolerant systems*. This first report from WP2 lists some of the key issues that we believe need to be resolved and provides brief descriptions of –and pointers to– relevant papers published by project members.

Contributors:

Many including people have written material for this deliverable (including the current editor); specific contributions include:

Section 2.2.2 written by Ian Oliver
Section 2.3.3 written by Victor Khomenko and Maciej Koutny
Section 2.3.4 written by Michael Butler
Section 2.4.1 written by Colin Snook
Section 2.4.3 written by Joey Coleman and Michael Butler
Chapter 3 written by Mike Poppleton
Chapter 4 written by Colin Snook
Chapter 5 written by Neil Evans
Chapter 6 written by Alex Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky
Chapter 7 written by Linas Laibinis and Elena Troubitsyna
Chapter 8 written by Alex Iliasov, Alexander Romaonvsky

Contents

1 Introduction **5**

2 Issues **6**
2.1 Formal methods in general . . . 6
2.1.1 Building a specification from parts . . . 6
2.1.2 Partial functions . . . 7
2.1.3 Role of invariants . . . 7
2.1.4 Controlling the order of operation execution . . . 8
2.2 Analysing systems . . . 8
2.2.1 Deriving the specification of control systems . . . 9
2.2.2 Domain modelling . . . 10
2.3 Issues concerned with concurrency . . . 11
2.3.1 Coping with interference . . . 11
2.3.2 Refining atomicity . . . 11
2.3.3 Process algebras and net theory . . . 12
2.3.4 Process Algebra and Event B . . . 13
2.4 Specific issues to do with fault tolerance . . . 14
2.4.1 Failure management . . . 14
2.4.2 Determining the failure specification of a system . . . 17
2.4.3 BPEL-like languages . . . 17
2.5 Role of support tools . . . 19
2.5.1 Synergy between model checking and reasoning . . . 19
2.5.2 Rigorous reasoning . . . 19
2.5.3 Role of Programming Languages . . . 19

3 Requirements Structure **20**
3.1 Introduction . . . 20
3.2 Failure Detection and Management for Engine Control . . . 20
3.3 Overview of methodology . . . 21
3.4 Conclusion . . . 22

4 Linking UML and B **24**
4.1 Motivation . . . 24
4.1.1 Building specifications from parts . . . 24
4.1.2 Choosing coherent useful abstractions . . . 25
4.1.3 Provide a more accessible notation for new users . . . 25

4.2 An Object-Oriented Approach 25
4.3 Overview of UML-B 26

5 Records in Event-B **28**
5.1 VDM Composites 28
5.2 A Property-Oriented approach in B 28
5.3 Refining Record Types 30
5.4 Combining Records (with Common Ancestors) 31
5.5 State using the Property-Oriented approach 31

6 Methodology for Formal Development of Mobile Location-Based Systems **33**
6.1 Introduction 33
6.2 System structure 34
6.3 Formal Development Process 35
6.3.1 Development of scopes and roles 35
6.3.2 Agent Design 36
6.3.3 B specification of the middleware 36
6.4 Discussion 36

7 Methodology for Formal Model-Driven Development of Communicating Systems **38**
7.1 Introduction 38
7.2 Pattern for specifying service component in B 38
7.3 Formal Service-Oriented Development 41

8 Exception Handling in Coordination-based Mobile Environments **46**
8.1 Exception Propagation through Coordination Space 46
8.2 Traps Mechanism 47
8.2.1 Discussion 48

9 The way ahead **49**
9.1 Link to Rodin Tasks 49
9.2 Addressing the issues in the next period 49
9.3 Relevant publications not cited above 50

Chapter 1

Introduction

Fault tolerance is a means for achieving dependability despite the likelihood that a system still contains faults and aiming to provide the required services in spite of them [ALLR04]. In our work we are focusing on developing methods supporting rigorous design of systems capable of tolerating faults in the system environment, faults of the individual components, component mismatches, as well as errors affecting several interacting components.

The objectives of Rodin are to contribute

1. methods for the formal specification and formal development of fault tolerant systems;
2. a tools platform and plugins which will support the use of such methods;
3. (five diverse) case studies which both drive the thinking and validate the other results.

The stated intention of the Rodin group is to make "Event-B" (a development of Jean-Raymond Abrial's "B" notation [Abr96]; see deliverables in WP-3 for current status) the prime exemplar of a suitable formal method.[1] In addition, members of the project have made significant contributions to Petri Net research [BDK01], VDM [Jon80, Jon90], Z [Abr85, Hay93] and UML [SB04, SOB04a]. The project is therefore in a unique position to explore issues around fault-tolerance, formal methods and tool support — this exploration will both be informed by and feed into the case studies.

This "Month 12" deliverable is the first summary of the work within the Rodin project on such methodological issues.

Chapter 2 recaps a number of important questions which relate to formal methods themselves; Chapters 3–8 develop specific points where progress has already been made; Chapter 9 indicates where we will next concentrate our efforts.

[1] Because of the parallel evolution of documents over this first year of the Rodin project, it is obviously not possible to have all of the examples in this deliverable comply to the description of "Event-B" in D7. This issue will be addressed in future deliverables as the tools become available.

Chapter 2

Issues

This chapter outlines a number of issues[1] which are relevant to the sort of methodology with which Rodin is concerned. Section 2.1 identifies questions which carry over from any formal method where we feel it worth considering what impact the application area of Rodin (fault-tolerant systems) might have; Section 2.3 repeats this pattern for concurrent systems since concurrency is bound to be a major aspect of many systems of interest; Section 2.4 is concerned more specifically with fault-tolerant systems; Section 2.2 (its order is determined so that material is explained only once) addresses issues concerned with getting the initial specification of a system; and Section 2.5 discusses some of the implications on tools to support formal methods for fault-tolerant systems (there is of course a parallel WP on the specific Rodin tools).

2.1 Formal methods in general

This section covers a number of technical issues concerned with formal methods for specification and development.

2.1.1 Building a specification from parts

There are occasions where it is desirable (or at least it appears to be so) to build up a large specification from parts. The arguments for so doing need to be closely examined partly because there appears to be a need to "flatten" such specifications in order to develop from them. (An old paper which sets out a challenge problem is [FJ90]. This paper was productive in the sense that there were responses representing several methods. One of the lessons is that engineering taste in such decompositions is as important as the formal notation.)

One interest in this area comes from the markedly different approaches taken in VDM, Z and B.

- VDM initially [Jon80] offered no help in building up specifications; in [Jon90] a method of "operation quotation" was suggested which facilitated the promotion of an operation on one state –say A– to be used in operations on states which included components of type A; the VDM standard [ISO95] took broadly this position but VVSL [Mid93] (which

[1] No attempt is made to reproduce in full material that is written at greater length elsewhere; we simply cite source papers.

was used in the original Praxis project on CDIS — see [Hal96]) and VDM++ [FL98, FLM$^+$05] both developed further ways of building up specifications;

- early versions of Z[2] also struggled with the problem of promotion and a range of approaches were discussed. The eventual "schema calculus" which is seen today as standard Z [HJN94] is loved by some but seen by others as a very dangerous textual copying mechanism which is difficult to reconcile with proofs;[3] and
- B [Abr96] is, in many respects, closer to VDM than it is to (what most people cite as) Z: for example, B has a clear textual distinction between machines, operations, pre and post-conditions; furthermore, Abrial's current thoughts on Event-B question the wisdom of building up of specifications for components.

There is an argument that Z schemas are useful for *developing* a specification and that a VDM or B machine style is necessary as a basis of development from a specification.

One reason that this debate needs to be resumed is that the need to describe normal and exceptional behaviour is a telling example of structuring.

2.1.2 Partial functions

Partial functions and operators occur frequently in program/system specifications and it is therefore important to decide how to handle them in formal reasoning. Classical logic (FOPC) does not really cope with undefined terms.

A variety of approaches have been tried – see [Jon96b, Jon95] for a (biased?) listing.

Interestingly, Jean-Raymond Abrial and Cliff Jones favour somewhat different approaches. Abrial argues [AM02] that one needs to show that all applications of partial functions are used within their defined domain. Jones uses a non-standard logic known as the "Logic of Partial Functions" [BCJ84, CJ91, JM94]

Abrial and Jones discussed the issue again on Jones' May 2005 visit to ETH and agreed that the tooling question (cf. Section 2.5) could be decisive to the question of usability.

(There are interesting links to be investigated with the position taken by the UML group on OCL.)

2.1.3 Role of invariants

The position taken in early VDM (cf. [Jon80]) was that data type invariants had to be implied by post conditions. This is very like the current interpretation in (Event-)B. After the first book, Jones switched to viewing objects with invariants as restricted types: quantifying over the type implies that the invariant is satisfied.

This issue is worth considering again, and as with partial functions, tool support is likely to be influential. (There are again interesting links to be investigated with the position taken by the UML group on OCL.)

[2] This is "pre-Z" to most people and refers to the time when Abrial and Jones were both in PRG at Oxford.

[3] There was a relevant debate (which represented all of the points of view) in the Helsinki plenary Rodin meeting between Abrial/Hall/Jones on exactly this topic.

2.1.4 Controlling the order of operation execution

VDM essentially did nothing explicit about controlling the order in which operations can be invoked (pre conditions are essentially proof obligations in that it is a mistake to (try to) invoke an operation when the pre condition is false).[4] The position of (original) B [Abr96] is similar. VDM++ [FL98, FLM^{+}05] embeds VDM ideas into an object-oriented (imperative) framework.

Event-B uses guarded statements in the style of "Action systems" [BvW98, Mor90].

A number of authors [But00, BL05, CSW03, AE04] have experimented with using process algebraic expressions to control which operations are "available" at any point in time. Jones has suggested that the two approaches be viewed as possible refinements (in either direction) of each other. He also suggests that the mapping from various OOLs to the pi-calculus suggests that a natural notation for creating instances of machines could be provided.[5]

2.2 Analysing systems

The notion of system itself needs careful analysis because we are clearly talking about a recursive notion of "systems of systems". Fortunately, the earlier FP-5 project DSoS produced a careful "conceptual model" [GIJ^{+}02].

The IST MATISSE project applied a system-level approach with B to the development of a railway signalling system. Part of this work is described in [But02]. The work described in [But02] is a system-level approach in the sense that is starts with an abstract formal model of the overall system to be controlled, including relevant control behaviour. This high-level model is then refined, adding detail about interactions between sub-components of the system, so that formal models may be decomposed into the system under control and the embedded controllers. Thus, rather than starting with formal models of the embedded controllers and verifying that they interact with the system under control, it starts with a model of the desirable behaviour of the overall system and uses refinement to derive models of the embedded controllers. The refinement is typically performed in several steps, and at each step it is verified that any behaviour of the refined model is allowed by the previous model, thus ensuring that the final detailed model is correct with respect to the original system-level model. The advantage of our approach is that it makes it easier to deal with the complexity of the designs. We start with a simple system-level model and introduce extra complexity one step at a time. The approach we take is particularly suited to systems which consist of multiple distributed controllers. In the system level model, we abstract away from any distribution and only introduce it in refinement steps.

The railway case study presented in [But02] is a simplified version of work that we are undertaking as part of the MATISSE project with input from one of the MATISSE partners, Siemens Transportation Systems. The system-level model describes features of the railway such as connections between track sections, occupancy of track sections by trains, basic movement of trains and changing of track connections. The model represents safety features only and does not represent mission-oriented features such as scheduling and routing. The paper

[4] This is actually a slight over-simplification in that the part of VDM more commonly associated with programming language semantics does have a set of sequencing "combinators" — but these are not normally mixed with pre/post conditions.

[5] An extended abstract is at [Jon05] the material has also been presented to IFIP WG2.3.

outlines a refinement which allows the system to be decomposed into the track behaviour and the train behaviour, with communication between the track and the trains.

2.2.1 Deriving the specification of control systems

The work (initially done in the DIRC project — see www.dirc.org.uk) on deriving specifications of control systems is extremely relevant to Rodin's objectives. The initial publication was [HJJ03] but the research has been presented widely and is being actively pursued within Rodin by Joey Coleman. Coleman and Jones have a joint paper at the forthcoming (Rodin-organised) REFT workshop. Because those proceedings are not yet available in print,[6] [7] we take the liberty below of extracting from the paper an overview of the "Hayes/Jackson/Jones approach".

The general idea of the "Hayes/Jackson/Jones" approach [HJJ03] is simple: for many technical systems it is easier to derive their specification from one of a wider system in which physical phenomena are measurable. Even though the computer cannot affect the physical world directly, it is still worthwhile to start with the wider system. The message can be stated negatively: don't jump into specifying the digital system in isolation. If one starts by recording the requirements of the wider (physical) system, the specification of the technical components can then be *derived* from that of the overall system; assumptions about the physical components are recorded as rely-conditions for the technical components.

In order to be able to write the necessary specifications, some technical work derived from earlier publications of Hayes, Jackson and Jones has to be brought together. The process of deriving the specification of the software system involves recording assumptions about the non-software components. These assumptions are recorded as rely conditions because we know how to reason about them from earlier work on concurrency (e.g. [Jon81, Jon83a, Jon96a]). In most cases, we need to reason about the continuous behaviour of physical variables like altitude: earlier work by Hayes (and his PhD student Mahony) provides suitable notation [MH91]. The emphasis on "problem frames" comes from Jackson's publications [Jac00].

A trivial example of the HJJ approach is a computer-controlled temperature system: one should not start by specifying the digital controller; an initial specification in terms of the actual temperature should be written; in order to derive the specification of the control system, one needs to record assumptions (rely conditions) about the accuracy of sensors; there will also be assumptions about the fact that setting digital switches results in a change in temperature. Once the specification of the control system has been determined, its design and code can be created as a separate exercise. At all stages — but particularly before deployment — someone has to make the decision that the rely conditions are in accordance with the available equipment. Figure 2.1 gives an abstract view of the HJJ approach. The referenced [HJJ03] outlines this procedure for a "sluice gate" controller. The analysis includes looking at tolerating faults by describing weaker guarantees in the presence of weaker rely conditions.

Notice that it is not necessary to build a complete model of the physical components like motors, sensors and relays: only to record assumptions. But even in the simple sluice gate example of [HJJ03], it becomes clear that choosing the perimeter of the system is a crucial

[6] Last minute addition: proceedings now available as Technical Report: [BJRT05]

[7] Ian Hayes gave an invited talk on the approach at REFT-05 and Jones gave an invited talk at DSVIS-05 on possible extensions to the approach.

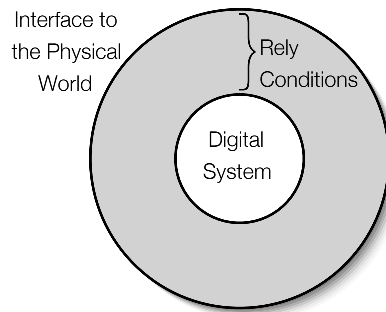


Figure 2.1: Bridging from the physical world to a digital control system

question: one can consider the physical phenomena to be controlled as the height of the gate, or the amount of water flowing; or the humidity of the soil; or even the farm profits. Each such scope results in different sorts of rely-conditions.

2.2.2 Domain modelling

Before describing what domain modelling is we first define the notion of a domain. There are many definitions of what a domain is but all have a commonality. We choose the definitions provided in [Eva04], [SM92, Old02] for the basis of this description.

> A domain defines a particular level of abstraction and the areas of concern for a description which is then presented in terms of a formal model using some suitable language (B, UML etc).

In [Old02] a domain is described as an area in which an ”Enterprise” conducts its business; how this is expressed and what information is contained in any description of a domain (i.e. a domain model) depends upon the methods used to describe that information. In [Eva04] a domain is defined as “a sphere of knowledge, influence or activity”. Domain modelling is thus the act of describing the captured knowledge about some domain.

The Shlaer-Mellor method [SM92] provides a structure about how domains interact in the context of some development project. In [RFW+04] this is made explicit by defining four categories of domains:

- Application
- Service
- Architecture
- Implementation

These terms refer to the rôle that a domain model is *playing* at any point in time with respect to a given development. One project might create something that is considered to be an application, while in another project this application is considered to be a service or architecture.

A **application domain** contains models of the "real world" for the purpose of describing the system under consideration from the users' point of view. The models in this domain will use the terms and concepts of the users only.

A **service domain** contains the basic services which is independent of the application domain for the support of the application domain. Such services might be persistence, fault tolerance facilities etc.

The **architecture domain** is the domain that provides the execution environment for all the application and service domains that have been described. The use of the term execution here does not necessarily refer to any machine level implementation but rather some environment that allows some form of execution [HJ89, Fuc92, GH96].

Finally the **implementation domain** is reached in which is defined and modelled any pre-existing components, operating system environments, programming languages, storage etc.

One might argue in the case of MDA as defined by the OMG [8] that the purpose of the MDA is to take (platform independent) models from the application domain and map them into models in a concretised application domain (platform specific) via some platform which is defined from combinations of artifacts from the service, architecture and implementation domains which together define the set of platforms that the system may be mapped onto.

2.3 Issues concerned with concurrency

Concurrency arises both in implementations where the designer is striving for performance by deploying many processors and is inherent in some problems. The latter case is commonly true where there is an interface to the physical world (e.g. handling many ATMs for a bank or processing signals from sensors which are obtaining input in real time).

The Rodin methods must therefore tackle the specification and design of concurrent systems. This section outlines some relevant research which will be re-evaluated within the Rodin context.

2.3.1 Coping with interference

Work on specifying and reasoning about the interference which is inherent with concurrency is addressed in many theses and papers. In particular, Jones' rely/guarantee research is addressed in [Jon81, Jon83a, Stø90, Col94, Xu92, Din00, BS01] The best (encyclopedic) overview is [dR01] but there is a clear need for a more usable survey of rely/guarantee-conditions and Coleman/Jones hope to tackle this before the Month 24 report.

2.3.2 Refining atomicity

The notion of "atomicity" is at the heart of our understanding of concurrent execution (see, for example, [Dij82, BJ05]) and links intimately with questions of fault tolerance [XRR+99]. (Other relevant topics include atomicity and formal refinement, atomicity and error isolation, atomicity and error recovery, atomicity and system structuring, atomicity and stepwise system development.)

[8] See www.omg.org/mda

A Schloß Dagstuhl event was organised in April 2004 on the topic of "Atomicity" (Romanovsky and Jones were two of the organisers of the event — several other members of the Rodin team attended the exciting event).[9] An overview reflecting the progress in bringing the diverse views together has been published in the specialist journals of several of the disciplines present (e.g. [JLRW05]); a special edition of the JUCS journal in July 2005 contains a selection of the papers which were presented.

The Rodin group sees this sort of event as an important way both to disseminate Rodin results and to make progress on a ubiquitous research question. They are planning a further Schloß Dagstuhl event on the topic which will be held in March 2006.[10] There will again be an international list of participants (the other co-organisers are from Seattle and Germany) and a mixture of research interests from databases, fault tolerance, machine architectures and theory.

Jon Burton (Newcastle) –together with Jones– have just (July 2005) begun research on a UK-funded (EPSRC) project on "Splitting (software) atoms safely". This project could provide useful input to Rodin work on development methods (see [Jon03] for the motivation of this research).

2.3.3 Process algebras and net theory

Petri nets [Rei85] and process algebras, e.g. [Hoa85, Mil89], are two of most successful formal models for the description and analysis of concurrent and distributed systems. They can be viewed as concurrent extensions of respectively finite automata and regular expressions.

Process algebras allow one to specify and reason *compositionally* about complex concurrent computing systems, by employing specific algebraic operators corresponding to commonly used programming constructs. Petri nets, on the other hand, provide both a *graphical representation* of such systems and, through employing the theory of partial orders and unfoldings [Kho03], efficient ways of *verifying* their correctness as well as expressing properties related to causality and concurrency in system behaviour. A decade ago the standard treatment of the structure and semantics of concurrent systems provided by these two types of models was different, making it virtually impossible to take full advantage of their relative strengths (i.e., compositionality and explicit asynchrony) when used in isolation. Since then the situation has changed considerably, in particular, due to the development of the Petri Net Algebra [BDK01] which is a generic model that embodies both Petri nets and process algebras, and thus directly supports explicit asynchrony and compositionality. For a carefully chosen (yet flexible) algebra of process expressions it provides corresponding well-behaved process-algebraic composition operators on nets. The resulting model is expressive enough to model process algebras (e.g., CCS and CSP [Hoa85, Mil89]), within a setup provided by two consistent semantics, respectively based on Petri nets and SOS rules in the style of [Plo81a]. One of the fundamental achievements of the Petri Net Algebra was a full consistency between two seemingly different semantics of process expressions: one based on suitably defined operational semantics rules, and the other derived from the standard Petri net semantics and compositional mapping from expressions to nets. Such a framework has been developed mainly for a net model based on the basic class of nets (1-safe Place/Transition nets). However, several results have been lifted to the high-level net framework and a concurrent specification language implemented in the PEP

[9]See http://www.dagstuhl.de/04181/

[10]See http://www.dagstuhl.de/06121/

toolkit [GB96]. From the point of view of Rodin, a crucial step was the development of an extension of the Petri Net Algebra [DKK04] capable of modelling the π-calculus [MPW92]. This has paved the way for the ongoing effort to construct a combined Petri net and process algebra model for dealing with *mobility* in concurrent systems based on the KLAIM process algebra [BBN$^+$03]. As a result, one should be able to apply Petri net based verification techniques to deal with mobile distributed systems, as described next.

Mobile systems are highly concurrent causing thus a *state space explosion* [Val98] during verification. One should therefore use approach which alleviates this problem; in our case, we employ the partial order semantics of concurrency and the corresponding Petri net unfoldings.

A *finite and complete unfolding prefix* of a Petri net $\mathcal{N}$ is a finite acyclic net which implicitly represents all the reachable states of $\mathcal{N}$ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* $\mathcal{N}$, by successive firings of transition, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated. If $\mathcal{N}$ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix.

Efficient algorithms exist for building such prefixes [Kho03], and complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, 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.

2.3.4 Process Algebra and Event B

In the Event-B approach, a system 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 [Hoa85]. CSP provides operators such as sequential composition, choice and parallel composition of processes, as well as synchronous communication between parallel processes.

The paper [BL05] describes an approach to using 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, the paper takes the view that the execution of an operation in a B machine corresponds to an event in CSP terms. Semantically a B machine is viewed 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. The approach described in [BL05] has been

implemented as part of the ProB model checker. ProB supports refinement checking between B models and between combinations of CSP and B.

There is much existing work on combining state based approaches such as B with process algebras such as CSP. The csp2B tool [But00] allows specifications to be written in a combination of CSP and B by compiling the CSP to a pure B representation which can be analysed by a standard B tool. The CSP support by csp2B is more restricted than that supported by PROB: csp2B does not support internal choice and allows parallel composition only at the outermost level unlike the arbitrary combination of CSP operators supported by PROB. The the CSP||B approach of [ST04] is focused on a style of combining CSP and B where the B machines are passive and all the coordination is provided by the CSP. This means the operations of their B machines cannot be guarded though they can have preconditions. They have developed compositional rules for proving that CSP controllers do not lead to violation of operation preconditions.

There has been much work on combining CSP with Z and Object-Z, including [Fis97], [Smi97] and [MD98]. Like our approach, these treat Z specifications as CSP processes and model the composition of the CSP and Z parts as parallel composition. The work described in [MS01] describes an approach to translating Z to CSP so that CSP-Z specifications can be model checked. This translation is not automated though. The Circus language is a rich combination of Z and CSP allowing Z to be easily embedded in CSP specifications and providing refinement rules for development [WC02]. We are not aware of any tools that allow for model checking of Z and CSP specifications directly.

2.4 Specific issues to do with fault tolerance

The Rodin group is, as mentioned above, striving to make fault tolerance part of the rigorous development process. [11]

(Material below in Chapter 6 should also be considered under this heading.)

2.4.1 Failure management

Critical control systems need to be able to cater for and manage failure of their sources of input information.[12] The role of failure management in an embedded control system is shown in Figure 2.2.

Inputs may be tested for signal magnitude and/or rate of change being within defined bounds, and for consistency with other inputs. If no failure is detected, some input values are passed on to the control subsystem; others are only used for failure detection and management. When a failure is detected it is managed in order to maintain a usable set of input values for the control subsystem. This may involve substituting values, and taking alternative actions. To prevent over-reaction to isolated transient values, a failed condition must persist for a period of time before a failure is confirmed. Once a failure is confirmed, more permanent actions are taken such as switching to an alternative source, altering or degrading the method of control, engaging a backup system or freezing the controlled equipment.

[11] The REFT-05 Workshop [BJRT05] had this aim.

[12] See for example the engine control case study.

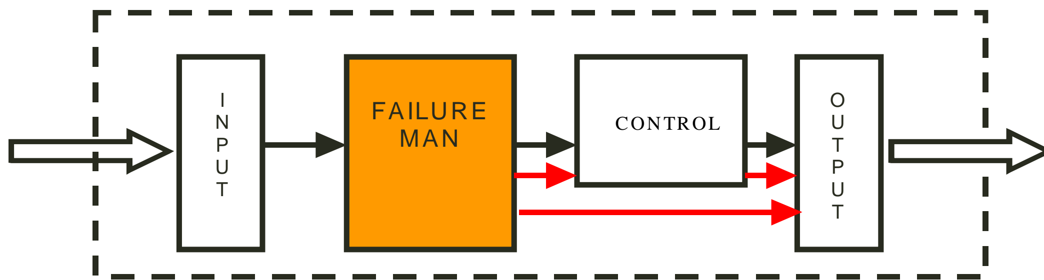


Figure 2.2: Context diagram for failure management subsystem

The nature of failure management is that different control actions and behaviour are required, dependent on the outcome of conditional logic for each of many inputs; this can result in complex overall behaviour. As a result it becomes difficult to identify reusable generic components that closely map to the problem domain. One approach is to model a failure management subsystem using UML-B. The flexibility of the object-oriented structures will improve reconfigurability and the use of formal verification will be particularly suited to safety critical applications. Modelling functional behaviour will provide the ontology to convey functional understanding and, through formal techniques, provide a way to map this to the code, reducing the semantic gap.

UML-B model of failure management

As an example of using UML-B to develop failure management systems we developed a simplified model and its verification. This was presented [SBEJ04b] at the 3rd International Workshop on Critical Systems Development with UML (CSDUML'04). Our first abstract model captures the overall states of the system. In subsequent refinements we model the stages in confirming a failure, the mechanism for freezing the system and the relationship between individual inputs and the collective state of the system. In these early stages we leave many aspects of the system under specified, saying only, for example, that an input may be detected as an unconfirmed failure and then may either recover or become a confirmed failure; but saying nothing about how or why these choices are made. Despite this (non-deterministic) underspecification the model embodies important properties about the interaction of the states of inputs that we verify by proof. To simplify the example we only considered input failures for which the controller freezes.

This first abstract model of failure management considers the overall state of the system. It defines the three main states of the controller in response to input validity conditions, which are; a) normal operation, b) frozen while attempting to confirm a possible input error, and c) hardfaulted when an input error has been confirmed. Note that once the system has hard faulted no further events may occur (the model is intentionally deadlocked).

In this first refinement we recognise that the system state is actually an abstraction of the states of many instances of input failure management (Figures 2.4 and 2.5). Each input has three possible states; ok, supect, and confirmed. Each input can have a good event (corresponding to a valid input value being detected) or a bad event (when an invalid value is detected). Some of these good and bad events (depending on the state of the full collection of inputs) refine the freeze, unfreeze and hardfault events from the abstract model. These are first_bad (the

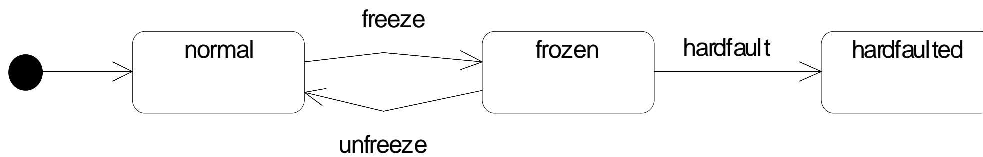


Figure 2.3: Statechart diagram of the abstract machine

first input to enter the suspect state), last_good and confirm respectively. When an input has confirmed detection of an invalid value, a guard on each transition prevents further events from being enabled. This models the intentional deadlock in the abstract model. The refinement relation gives the correspondence between the equivalent states of the two models. The system is normal when no inputs have detected invalid values, frozen when at least one input has detected an invalid value but not confirmed, and hardfaulted when an input has confirmed an invalid value. Note that we use a 'Petri' style interpretation of the state model (where each state is a variable whose value is the set of instances in that state) since this makes it easier to specify the collective state of the class in transition guards. Verification proves that the collective state of the inputs behaves in accordance with the overall system states; normal, frozen and hardfaulted.

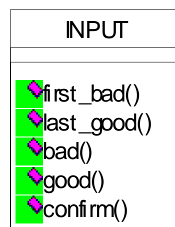


Figure 2.4: Class diagram of the first refinement

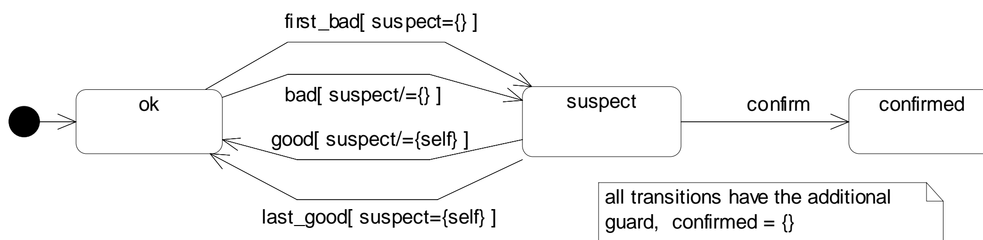


Figure 2.5: Statechart diagram of the INPUT class

Further details and levels of refinement can be found in the paper. In later refinements we find that each input may consist of several different tests and that tests need to be confirmed by some counter mechanism. Later still, we begin to distinguish the different kinds of tests using class specialisation. The use of UML-B in this way combines the UML class structuring

of the specification with the B refinement method. The most abstract levels of model embody essential properties that the system must fulfil, whatever its detailed specification eventually is.

In other work on requirements engineering of failure management systems (see Chapter 3) we have identified a similar role for requirements 'rationale'. In future work we plan to investigate ways to integrate the abstract modelling described above with our work on requirements engineering.

2.4.2 Determining the failure specification of a system

Section 2.2.1 has outlined the "Hayes/Jackson/Jones" approach as it applies to idealised physical components such as sensors and motors. The cited paper [HJJ03] also addresses how to look at failures as weaker assumptions on the environment. Neatly combining the idealised and fault tolerant specifications is currently being written up for a journal paper (by the original three authors).

2.4.3 BPEL-like languages

One of the current active areas in web services is the design of choreography and orchestration languages to coordinate activities across a distributed set of web service providers. The intended typical applications tend to centre on e-commerce, but bio-informatics and ambient intelligence research also provides the occasional example.

Among the major efforts is BPEL[13], for which the last official specification was published in May 2003 [ACD+03]. The language includes a compensation feature that, while it has good points, has some arbitrarily imposed flaws. An operational semantics of a subset of BPEL — including its compensation mechanism — can be found in [Col04].

BPEL introduces compensation as a tool for error handling during long-running transactions so that actions can be reversed after their completion. This is in contrast to BPEL's regular fault handling mechanism, which deals with cleaning up after actions that are currently in progress. However, the language specification arbitrarily restricts the invocation of compensation to be only within the context of a fault handler. This has the pragmatic implication that compensation can only be used for error-handling.

Unfortunately, this is an extremely narrow view of what compensation can be used to achieve. One of the fundamental differences between long-running transactions and short-running (ACID) transactions is the nature of change in the environment. ACID transactions have the implicit assumption that environmental changes are minor due to their short duration. Long-running transactions, however, cannot maintain this assumption: even the conditions which initiated the transaction can be wholly or partially negated (and restored) over the lifetime of the transaction.

Compensation, in general, is a mechanism that can be used to handle at least some of the effects of a radically changing environment. Compensation allows a process to reverse the effects of an action without having to literally restore the previous state as a database rollback would. The obvious extension of this is to allow compensation to change a process' state such that some integrity constraint is satisfied. This would help the process adapt to environments

[13] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`

where the initial requirements for the transaction are likely to change over time, or are even incomplete at the start of a transaction.

A fairly simple argument is made in [Col05] that the restriction on the use of compensation in BPEL be removed. The core of the argument, as described above, is that it would give the language far more flexibility to handle changes in the process' environment. It should be noted that some modifications have been made to the specific structure of BPEL's compensation construct in recent drafts of the language specification, but the restriction has remained unchanged.

The Compensating CSP (cCSP) language was introduced by Butler et al. [BHF04] as a language to model long running transactions in the framework of CSP process algebra [Hoa85]. The semantics of the cCSP language was described by using denotational semantics (trace semantics). The paper constructs a model of long-running transactions within the framework of the CSP process algebra, showing how the compensations are orchestrated to achieve the illusion of atomicity. It introduces a method for declaring that a process is a transaction, and for declaring a compensation for it in case it needs to be rolled back after it has committed. The familiar operator of sequential composition is redefined to ensure that all necessary compensations will be called in the right order if a later failure makes this necessary. The techniques are designed to work well in a highly concurrent and distributed setting. In addition we define an angelic choice operation, implemented by speculative execution of alternatives; its judicious use can improve responsiveness of a system in the face of the unpredictable latencies of remote communication. Many of the familiar properties of process algebra are preserved by these new definitions, on reasonable assumptions of the correctness and independence of the programmer-declared compensations.

Previously one of the authors (Butler) was involved in the development of the StAC (Structured Activity Compensation) language [BF00, BF04] for modelling long-running business transactions which includes compensation constructs. An important difference between StAC and cCSP is that instead of the execution of compensations being part of the definition of a transaction block, StAC has explicit primitives for running or discarding installed compensations (*reverse* and *accept* respectively). This separation of the *accept* and *reverse* operators from compensation scoping prevents the definition of a simple compositional semantics: the semantics of the reverse operator cannot be defined on its own as its behaviour depends on the context in which it is called. This necessitated the use of configurations involving installed compensation contexts in the operational semantics for StAC. Note that BPEL also has an operator for explicit invocation of compensation. A mapping from BPEL to StAC may be found in [BFN05].

Bruni et al. [BMM05] have developed an operational semantics for a language with similar operators to cCSP, including compensation pairs and transaction blocks (or sagas as they call them). As in cCSP, and unlike StAC, the invocation of compensation in a saga is automatic depending on failure or success which leads to a neater operational semantics. However, unlike the work presented here, the operational semantics in [BMM05] is defined by using big-step semantics. Big-step semantics describe how the overall results of the execution are obtained. The big step semantics are closer to the trace semantics while our small-step semantics describes how compensating processes should be executed. A comparison of the operators of cCSP and the language described in [BMM05] may be found in [BBF+05].

2.5 Role of support tools

There is no intention that this section should repeat considerations from the relevant Rodin WP but there are some questions where tool considerations have a direct impact on methods. One example above is the technique for handling partial functions in logic (cf. Section 2.1.2). This section reviews some other questions of the same sort.

2.5.1 Synergy between model checking and reasoning

Development methods which are classed as "correctness by construction" (VDM and B are certainly in this camp) obviously need tool support and the existing "Atelier-B" tools are an outstanding example of tools which are used in industry. In this class of methods, the designer develops and then refines abstractions.

To make the contrast as sharp as possible, model-checking is initially described in its original form. A key advantage of model checking is that it can be applied without an initial specification. Problems like potential deadlock can be detected from the final code. The designer is not required to have recorded design abstractions. (There are of course limitations to this "free lunch" — but they are not the issue here.)

The state explosion problem discussed in Section 2.3.3 can be tackled by using abstractions. A key method question related to deriving synergy from correctness by construction and model checking is the extent to which design abstractions can be carried over to model checking. Tools could play a key role here.

2.5.2 Rigorous reasoning

Apart from the need for support tools which assist a designer to undertake completely formal "correctness by construction", there are interesting questions around what can be done to support "rigorous development". (Some of these issues have been discussed in [JJLM91].) The level of support which can be provided and, in particular, the support for tracing the impact of changes, will obviously have an affect on the methods proposed in Rodin.

2.5.3 Role of Programming Languages

Jean-Raymond Abrial has discussed (cf. Jones' May visit to ETH) the role of programming languages in a world where software is developed using "correctness by construction" techniques. Certainly, Abrial's considerable success in earlier projects (e.g. Paris Metro) suggest that the place of high-level languages will change. This could offer a resolution of the old question of whether one needs a (full) axiomatic semantics of some of the untidier programming languages in which many systems are constructed today.

Chapter 3

Requirements Structure

3.1 Introduction

The structuring of requirements, supported by some means of classification (via a *taxonomy* of requirements for the domain in question), has been recognized in the Rodin project as a significant early-lifecycle factor affecting *inter alia* the ease of modelling, design and productivity. The issue was introduced to the project by J.-R. Abrial at the Chilworth workshop, Dec. 2004, by means of an example structured requirements document [Abr04].

This early perspective influenced the work at Southampton on Case Study 2: "Failure Detection and Management for Engine Control". This chapter is a methodological view of work to date on Case Study 2, and constitutes a précis of our papers [SPJ05a, SPJ05b].

We consider the failure detection and management function for engine control systems as an application domain where product line engineering is indicated. The need to develop a generic requirement set - for subsequent system instantiation - is complicated by the addition of the high levels of verification demanded by this safety-critical domain, subject to avionics industry standards. Our early case study experience has suggested a candidate methodology for the engineering, validation and verification of generic requirements using domain engineering and Formal Methods techniques and tools.

3.2 Failure Detection and Management for Engine Control

The case study (Case Study 2 in Rodin), provided by AT Engine Controls of Portsmouth, is a failure manager which filters environmental inputs to the engine control system, providing the best information possible whilst determining whether a component has failed or not. The role of failure detection and management in an embedded control system is described in Section 2.4.1.

Our approach contributes to the failure detection and management domain by presenting a method for the engineering, validation and verification of generic requirements for product-line engineering purposes. The approach exploits genericity both *within* as well as *between* target system variants. Although product-line engineering has been applied in engine and flight control systems [Lam97, Fau00], we are not aware of any such work in the FDM domain.

Our approach contributes methodologically to product-line requirements engineering in its integration of informal domain analysis with domain engineering that exploits both UML and Formal Methods technology. We develop a generic requirements specification using a UML-B

model and a tabular data schema. We present the process of domain engineering, validating and verifying a generic model and an example instance model. The UML-B is translated to B with the U2B tool, and then verified by model-checking with ProB; this verifies both the generic requirement set and the system instance.

3.3 Overview of methodology

We give an overview of the prototype methodology - see Figure 3.1. The first stage is an informal domain analysis which is based on prior experience. A taxonomy of the kind of generic requirements found in the application domain is developed and, from this, a *first-cut* generic model is formed, in object-association terms, naming and relating the generic requirements.

The identification of a useful generic model is a difficult process and therefore further exploration of the model is warranted. This is done in the domain engineering stage where a more rigorous examination of the first-cut model is undertaken, using the B-method and the Southampton tools. In Southampton we have developed two tools to support formal system development in B: ProB and U2B. ProB [LB03] is a model checker for B. UML-B [SOB04a] is a profile of UML that defines a formal modelling notation using class diagrams and statecharts, and has a mapping to the B language. The U2B [SB04] translator converts UML-B models into B components.

The first-cut model is animated by creating typical instances of its generic requirement objects, to test when it is and is not consistent. This stage is model validation by animation, using the ProB and U2B tools, to show that it is capable of holding the kind of information that is found in the application domain. During this stage the relationships between the objects are likely to be adjusted as a better understanding of the domain is developed. This stage results in a *validated* generic model of requirements that can be instantiated for each new application.

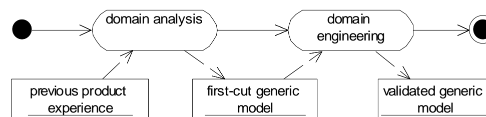


Figure 3.1: Process for obtaining the generic model

For each new application instance, the requirements are expressed as instances of the relevant generic requirement objects and their associations, in a tabular *instance* model - see Figure 3.2. The ProB model checker is then used to verify that the application is consistent with the relationship constraints embodied in the generic model. This stage, producing a verified *consistent* instance model, shows that the requirements are a consistent set of requirements for the domain. It does not, however, show that they are the right (desired) set of requirements, in terms of system behaviour that will result.

The final stage, therefore, is to add dynamic features to the instantiated model in the form of variables and operations that model the behaviour of the objects in the domain and to animate this behaviour so that the instantiated requirements can be validated. This final stage of the process - "validate instantiation" in Figure 3.2 - is work in progress.

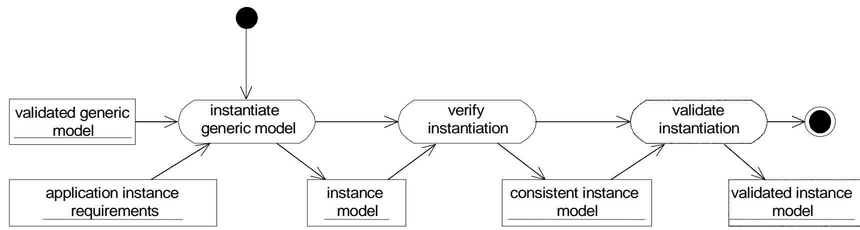


Figure 3.2: Process for using the generic model in a specific application

3.4 Conclusion

We have illustrated a product-line approach to the rigorous engineering, validation and verification of generic requirements for critical systems such as failure management and detection for engine control. The approach can be generalised to any relatively complex system component where repetitions of similar units indicate an opportunity for parameterised reuse but the extent of differences and interrelations between units makes this non-trivial to achieve. The product-line approach amortises the effort involved in formal validation and verification over many instance applications.

The methodology and tools presented are work in progress. During the domain analysis phase we found that considering the rationale for requirements revealed key issues, which are properties that an instantiated model should possess. At the moment, however, these are not enforced by the generic model. Key issues are higher level requirements that could be expressed at a more abstract level from which the (already validated) generic model is a refinement. The generic model could then be verified to satisfy the key issue properties by proof or model checking. This matter is considered in [SBEJ04a] which gives an example of refinement of UML-B models in the failure management domain. The domain analysis process of Figure 3.1 would then be elaborated as shown in Figure 3.3.

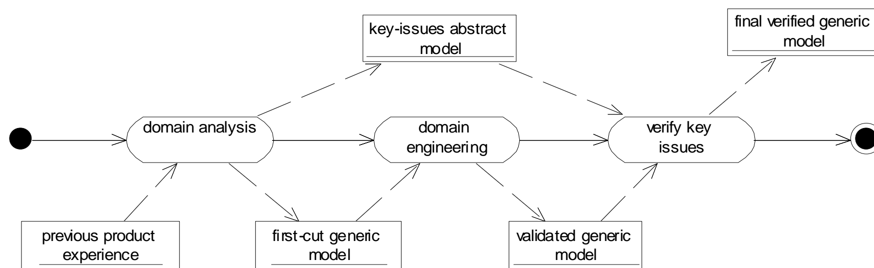


Figure 3.3: Elaboration of domain analysis process to show refinement of key issues

Further development is required to validate instance models, as per Figure 3.2. Whilst an instance model can be verified against the constraints embodied within the generic model (i.e. that it is a valid instantiation of the generic model), it may be the wrong configuration. That is, it may specify the wrong run-time behaviour. A further stage to validate the specific configuration is envisaged. Formal refinement could be utilised to add new variables and events

to represent the dynamic behaviour of the system. This would allow the specific configuration to be validated via animation.

We have indicated that requirements for support tools are emerging from the case study; such tools are being specified and planned. Simple database support is required for product line data definition: maintenance and evolution of the sets of requirement instance definition tables that define application product instances.

The work identified the need for finer-grained diagnosis of invariant violation in ProB: ProB could be enhanced to provide, for example, a data counterexample causing an invariant violation. A related need is for validation and debugging support for bulk data upload. Given that much of this validation is based on the object types and associations in the generic class model of the requirements, this is again a job for a database tool. We are also examining ways to feed such data counterexamples back to the UML diagram of the generic model; clearly a more user-friendly approach.

Chapter 4

Linking UML and B

The Rodin project will re-develop and enhance the UML-B profile [SOB04b], which is a profile of the UML that is precise and semantically well defined via equivalence to B. UML-B includes a constraint and action language, μB, that is derived from B.

4.1 Motivation

The methodological motivation for linking UML and B can be considered from two viewpoints. From the point of view of a typical UML user, linking with B brings greater rigour and precision to specifications and provides access to verification tools. However, since Rodin is primarily a formal methods development project we concentrate on this viewpoint. The main reasons for linking UML and B from this viewpoint are to:

1. build specifications from parts,
2. help choose coherent useful abstractions,
3. provide a more accessible notation for new users.

4.1.1 Building specifications from parts

UML-B brings the abstract data type approach, upon which object-orientation is based, to B specification. UML-B provides a mechanism to specify the state and associated behaviour of a kind of abstraction and then to apply (i.e. 'lift') that specification to an indeterminate set of abstractions that participate in a wider specification. Analogous facilities exist in Z (schema promotion) and VDM (Modules) but B lacks this facility. B has an encapsulation mechanism (machines) that allows variables to be grouped with the operations that act upon them. It is also possible, via machine renaming, to instantiate several instances of a machine. However, there is no mechanism to use the behaviour defined in this way to specify an indeterminate or variable set of instances. For example, in Z, 'promotion' enables schemas to be used to define a behaviour that is then promoted and bound to a set of instances at a higher level. This limitation is overcome in U2B by explicitly modelling the set of instances within the B and modelling each class feature with a function whose domain is the set of instances. UML-B adopts the UML state machine specification notation as a means of expressing the primary

state-transition behaviour of a specification part. Again, this can be applied to each instance kind and lifted to describe a set of such instances.

4.1.2 Choosing coherent useful abstractions[1]

The main difficulties in writing a formal specification are the need to commit to abstractions at an early stage and the difficulty of subsequently altering these abstractions. *Abstractions* are needed to achieve a *close mapping* between model concepts and the problem domain. *Progressive evaluation* validates the chosen abstractions before too much reliance is placed on them (*premature commitment*). Improved animators and model checkers would provide this exploratory evaluation. The problem is compounded by the difficulty of visualising abstractions in a mathematical notation. Graphical representation of formal models provides better *visibility* of abstractions and how they interact to compose the whole. This is valuable when assessing abstractions thereby alleviating premature commitment. A graphical design tool decreases *viscosity* (the effort of changing abstractions) since the diagrammatic symbols represent significant mathematical infrastructure and are therefore much quicker to re-arrange.

4.1.3 Provide a more accessible notation for new users

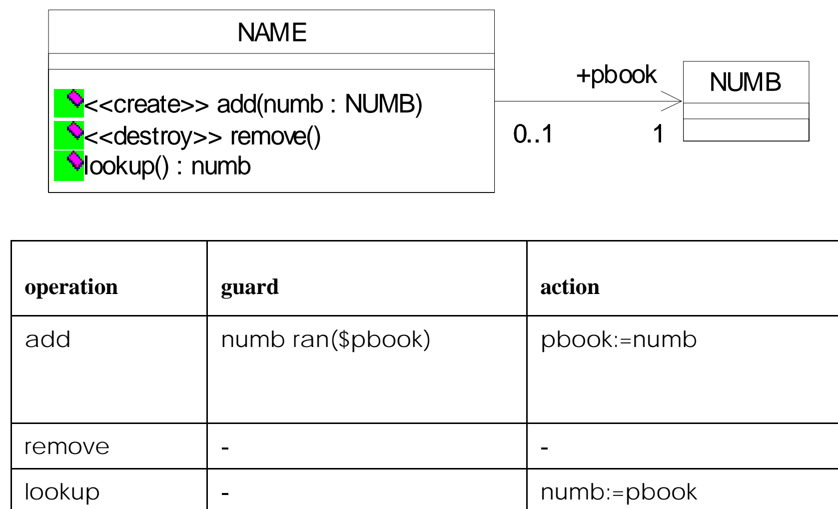
UML, despite not having a formal semantics has proved relatively popular in industry. By basing a formal notation on the UML, these users get a familiar feel during their introduction to the method. The automation of large amounts of 'infrastructure' B, represented by diagrammatic symbols, allows new users to concentrate on smaller additions of particular constraints and actions. This provides a good lead-in to formal specification in B. The diagrammatic specification also allows users to better visualise the specification for validation purposes. In a textual B specification it is sometimes possible to lose sight of the real-world mapping of mathematical constructs so that the specification, although perhaps consistent, may not specify what was intended.

4.2 An Object-Oriented Approach

The UML is based on an object-oriented modelling approach including the following key concepts: objects, encapsulation, class, generalisation/specialisation and messages. State machines can be used to model the behaviour of the objects. We utilise and deviate from these concepts as described below. (Definitions are from [RJB98]).

*An **object** is a concrete manifestation of an abstraction; an entity with a well-defined boundary and identity that encapsulates state and behaviour.* For our abstract systems modelling, objects are an abstraction of parts of a system that include state and events associated with those parts of the state. The state is modelled by attributes (variables of basic types), associations (variables whose type is based on a set of objects) and state machines. Events are defined by a guard (i.e., predicate on the state) that shows when they can occur and a substitution that shows how the state is changed by the event. For our modelling, object encapsulation is not important because the variables represent abstract state and we are only concerned with the

[1] The terms in italics in this paragraph are taken from the cognitive dimension framework [Gre89], which provides a broad-brush qualitative tool for assessing notations and interfaces.



operation	guard	action
add	numb ran($pbook)	pbook:=numb
remove	-	-
lookup	-	numb:=pbook

Figure 4.1: UML-B model of a telephone book

effect operations have on the state, not how they achieve it. Sometimes events affect the state of other objects and we indicate this by specifying the change of value directly or by using a 'subroutine' of the other object. A form of encapsulation is provided by 'Packages', which are used to modularise the model into, for example, sub-models and refinements.

*A **class** is a description of a set of objects that share the same attributes, operations, relationships and semantics.* We use classes to define sets of similar objects and 'specialisation/generalisation' relationships to show that the specialised class' objects are a subset of the generalised class' objects that have some additional or refined features.

*A **message** is a specification of a communication between objects...* Since we do not aim to 'modularise' a specification using encapsulation, message passing between objects is not applicable. The operations in our model represent events that occur spontaneously.

*A (behavioural) **state machine** is a behaviour that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.* We use state machines to model the conceptual state of an object which is additional to any state modelled by other variables of the object. The transitions represent the behaviour of events of the objects.

Modelling can be useful for many stages in the realisation of a system component. For example, [Dan02] classifies models into conceptual (analysis of the real world problem), specification (model of the requirements) and implementation (explanation of the built system). We envisage that UML-B could be used for any of these stages but currently we concentrate on conceptual modelling.

4.3 Overview of UML-B

To give a flavour of UML-B, consider the specification of the telephone book in Figure 4.1. The classes, NAME and NUMB represent people and telephone numbers respectively. The

association role, pbook, represents the link from each name to its corresponding telephone number. Multiplicities on this association ensure that each name has exactly one number and each number is associated with, at most, one name. The table shows μB conditions and actions for some of the operations. The add operation of class NAME has the stereotype $\ll$create$\gg$ which means that it adds a new name to the class. It takes a parameter numb, which must be an instance of the class, NUMB, but not already used in a link of the association pbook (see μB operation guard), and uses this as the link for the new instance (see μB operation action). The remove operation has no μB action; its only action is the implicit removal of self from the class NAME.

Chapter 5

Records in Event-B

Motivated by the need to specify the CDIS subset (CS4) in Event-B, we will have to represent some VDM constructs in B notation. Records (or *composites*) are used frequently in the CDIS subset, and VDM specifications in general, so it is worthwhile investigating how records (with arbitrary field types) can be defined and manipulated in B. More generally, it would be beneficial to incorporate some subtyping/inheritance-like properties of records to enable the reuse of such structures, and to allow better conceptual modelling during a development.

We begin with a brief overview of composites in VDM, and this is followed by a 'property-oriented' definition of record types in B notation. We propose some syntactic sugar to make such definitions more succinct.

5.1 VDM Composites

A composite type consists of a name followed by a list of component (field) names, each of which is accompanied by its type. In general, this looks like:

$$\begin{array}{lll} \mathit{type_name} & :: & \mathit{component_name}_1 : \mathit{component_type}_1 \\ & & \vdots \\ & & \mathit{component_name}_n : \mathit{component_type}_n \end{array}$$

It is possible to constrain the type by including an invariant for the values of the components.

State in VDM is declared as a special kind of record whose components are the state variables which can be accessed and modified via *operations* (functions having side effects on the state).

5.2 A Property-Oriented approach in B

This approach attempts to mimic the record type definitions of VDM by using the **SETS**, **CONSTANTS** and **PROPERTIES** clauses of B machines. One of the motivations of this work is to enable a stepwise development of complex record structures by introducing additional fields as and when they become necessary. This is akin to inheritance in object-oriented programming in which classes are *restricted* or *specialised* by introducing additional attributes. A coalgebraic view of classes and objects [Jac96] gives a natural semantics for inheritance.

CONTEXT *Func*
SETS *R* **;** *A* **;** *B*
CONSTANTS *r1* , *r2* , *set_r1* , *set_r2*
PROPERTIES

$r1 \in R \to A \wedge$
$r2 \in R \to B \wedge$

$set_r1 \in R \times A \to R \wedge$

$\forall (r , a) . (r \in R \wedge a \in A \Rightarrow r1 (set_r1 (r , a)) = a) \wedge$
$\forall (r , a) . (r \in R \wedge a \in A \Rightarrow r2 (set_r1 (r , a)) = r2 (r)) \wedge$

$set_r2 \in R \times B \to R \wedge$

$\forall (r , b) . (r \in R \wedge b \in B \Rightarrow r2 (set_r2 (r , b)) = b) \wedge$
$\forall (r , b) . (r \in R \wedge b \in B \Rightarrow r1 (set_r2 (r , b)) = r1 (r)) \wedge$

$r1 \otimes r2 \in R \twoheadrightarrow A \times B$
END

Figure 5.1: A simple record type

Using coalgebraic specification techniques, we present a B machine to model the composite type R, which is declared as follows:

$$\begin{array}{lll} R & :: & r1 : A \\ & & r2 : B \end{array}$$

That is, R is a record with two fields, $r1$ and $r2$, of type A and B respectively. In B, we begin by declaring three deferred sets R, A and B in the **SETS** clause. This is shown in Figure 5.1. The sets A and B correspond to the types A and B in the declaration and, as such, these could be interpreted as specific B types (such as *NAT*), or could themselves be record types. However, the set R represents the record type that we are trying to specify. This remains deferred until we are sure that we do not want to refine it by introducing additional fields. Instead, we specify properties of the set within the **PROPERTIES** clause.

Two *accessor* functions are declared in the **CONSTANTS** clause to retrieve the fields of a record instance: $r1$ retrieves the value of the field of type A, and $r2$ retrieves the value of the field of type B. In addition, we have declared two functions that update these fields. That is, given a record instance r and an element a of type A, $set_r1(r,a)$ returns a new record whose $r1$ field has value a. Similarly, $set_r2(r,b)$ updates the $r2$ field.

The properties of these functions are also defined in the **PROPERTIES** clause. In particular, note that the *set* functions do not change the other fields, and for every pair of values from A and B there is a record instance whose fields have these values. (This is expressed succinctly using the $\otimes$ (direct product) operator.) It is possible to infer other properties of the observable values, such as the commutativity of different *set* functions: $ri(set_rn(set_rm(r,x),y)) =$

CONTEXT *FuncR*
REFINES *Func*
SETS *C*
CONSTANTS *Q* , *r3* , *set_r3*
PROPERTIES

$Q \subseteq R \wedge$

$r3 \in Q \rightarrow C \wedge$

$set_r3 \in Q \times C \rightarrow Q \wedge$

$set_r1 [Q \times A] \subseteq Q \wedge$
$set_r2 [Q \times B] \subseteq Q \wedge$

$\forall (r , a) . (r \in Q \wedge a \in A \Rightarrow r3 (set_r1 (r , a)) = r3 (r)) \wedge$
$\forall (r , b) . (r \in Q \wedge b \in B \Rightarrow r3 (set_r2 (r , b)) = r3 (r)) \wedge$
$\forall (r , c) . (r \in Q \wedge c \in C \Rightarrow r3 (set_r3 (r , c)) = c) \wedge$
$\forall (r , c) . (r \in Q \wedge c \in C \Rightarrow r1 (set_r3 (r , c)) = r1 (r)) \wedge$
$\forall (r , c) . (r \in Q \wedge c \in C \Rightarrow r2 (set_r3 (r , c)) = r2 (r)) \wedge$

$(r1 \otimes r2 \otimes r3) [Q] = A \times B \times C$
END

Figure 5.2: A restricted record type

$ri(set_rm(set_rn(r,y),x))$ if $n \neq m$, for all i (i.e. they are indistinguishable from the user's point of view).

We propose an introduction of composite-like syntax in B, from which it would be possible to automatically infer the sets, constants and properties necessary to declare record types, such as those given in Figure 5.1. We also propose some syntactic sugar to alleviate the complexity of multiple field updates. For example, instead of the expression $set_r1(set_r2(r,x),y)$ we could have $r \oplus \{r1 \mapsto y, r2 \mapsto x\}$.

5.3 Refining Record Types

We now investigate the effect of introducing a new accessor function and *set* function to *R*. This is shown in Figure 5.2. The aim is to model a restricted version of *R*, which we name *Q*, in which a third field $r3$ has been introduced. We propose a syntax for this as follows:

$Q\ extends\ R \quad :: \quad r3 : C$

Notice that this is a refinement of the machine given in Figure 5.1. Hence, the properties declared in the refinement are in addition to those of the original machine. These include the effect of set_r1 and set_r2 on the new accessor function $r3$, and the effect of set_r3 on the

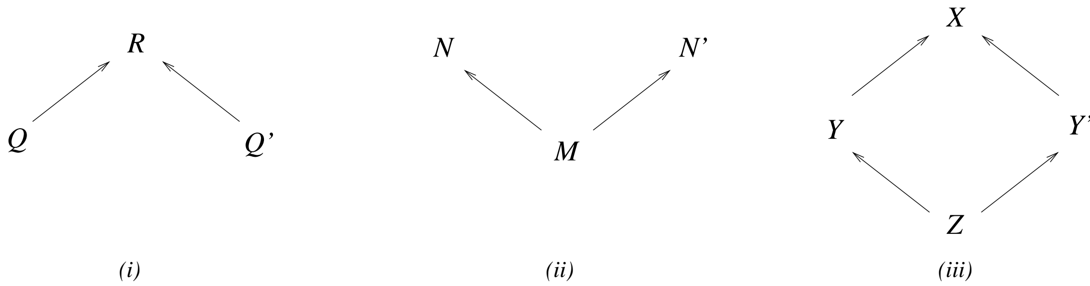


Figure 5.3: Possible Record Hierarchies

accessors $r1$ and $r2$. In addition, note that we have strengthened the requirements on set_r1 and set_r2 so that, however these functions are implemented, they must only update records of type Q by returning records of type Q (otherwise, the first two universally quantified properties of Figure 5.2 would not make sense). The final property states that all possible field combinations are available in Q.

5.4 Combining Records (with Common Ancestors)

In addition to a single chain of refinements, the approach to extending record types presented above permits other, less restrictive, kinds of development. The diagrams shown in Figure 5.3 give three possible extension hierarchies. Each of these is meaningful in a coalgebraic setting. Hence, we do not constrain the structuring of **CONTEXT** machines. In (i), two different extensions, Q and Q', extend the same record type R. That is, Q and Q' have a common ancestor. In (ii), the record type M combines the record types N and N'. Since N and N' have no common ancestors, it is important that there are no clashes in their field names. Intuitively, in M, we expect the fields of N' to remain unaffected by the set operations of N, and vice versa. Constraints are added to the **PROPERTIES** clause of M to enforce this. Diagram (iii) in Figure 5.3 is a combination of (i) and (ii).

5.5 State using the Property-Oriented approach

A state variable that is declared to be of type R holds a record instance of that type. Its components can be observed by using the accessor functions, and component modifications can be made via the *set* functions. Within the **OPERATIONS** clause, it is possible to define operations that modify these variables directly (rather than passing records to, and returning records from the functions declared in the **CONSTANTS** clause).

In order to allow additional fields to be added to a record type (via refinement), operations that update the existing fields of a variable are defined using an **ANY** clause. This means any 'hidden' fields (i.e. those fields that will be revealed later through refinement) are assigned non-deterministically. Refined operations can then replace this non-determinism with a deterministic assignment. For example, consider a variable r that is declared to be of type R given in Figure 5.1. An operation to update the $r1$ field of r could be defined as in Figure 5.4.

```
Update_r1_of_r ( x ) ≙
 PRE  x ∈ A THEN
   ANY  y WHERE
     y ∈ R ∧
     r1 ( y ) = x ∧
     r2 ( y ) = r2 ( r )
   THEN
     r := y
   END
 END
```

Figure 5.4: An update operation

In the context of the simple definition of *R* (Figure 5.1), this operation is equivalent to the substitution $r := set_r1(r,x)$. In a context that defines *r* (and quantified variable *y*) to be of the refined type *Q* given in Figure 5.2, then this operation would be weaker than the function set_r1 because the new field $r3$ of *r* would be assigned non-deterministically. Of course, the operation could be refined too in order to assign something meaningful to this field.

Chapter 6

Methodology for Formal Development of Mobile Location-Based Systems

6.1 Introduction

The agent technology naturally solves the problem of decoupling complex software into smaller parts that are easier to design, code and maintain. It helps to use distributed computing power effectively while hiding many of the details and complexities of a hosting environment. Agent software is designed to interact with other agents during its lifetime. To make full use of agent communication and migration capabilities, we need to assume systems are composed dynamically out of agents developed independently at different sites and for different purposes. Such configurations are impossible if agents are merely anonymous black-boxes. Hence we need to create a formal development methodology that would ensure interoperability of independently designed agents and correctness of the mobile system. During the first year of RODIN we focused our efforts on establishing the basis for achieving this.

In our view, to cooperate, agents must be based upon some common specification of their functionality. This specification should be formally developed and verified to ensure the properties of the application composed of agents. Developers of individual agents can independently extend the specification (using a refinement method) to add unique features without losing compatibility with other agents derived from the same specification.

Our work is based on an asymmetric model of the agent system within the *location-based* paradigm introduced in [IR05]. The asymmetric scheme is closer to the traditional service provision architectures. It can support large-scale mobile agent networks in a very predictable and reliable manner. Moreover, location-based architecture eliminates the need for employing complex distributed algorithms or any kind of remote access. This allows us to guarantee atomicity of certain operations without sacrificing performance and usability. This scheme also provides a natural way of introducing context-aware computing by defining location as a context. The main disadvantage of the location-based scheme is that an additional infrastructure is always required to support mobile agent collaboration.

Our *coordination paradigm* is based on Linda [Gel85] - the dominating environment in which a number of mobile systems are built (including Lime [PMR99], Klaim [NFP98], etc.). Linda is a set of language-independent coordination primitives that can be used for communication and coordination between several independent pieces of software. Linda-based co-

ordination systems support both physical mobility, such as a device with running application travelling along with its user across network boundaries, and logical mobility, when a software application changes its hosting environment. In the rest of the chapter we introduce the basic elements of the CAMA (context-aware mobile agents) methodology for formal development of mobile systems.

6.2 System structure

The CAMA (context-aware mobile agents) system consists of a set of *locations*. Active entities of the system are *agents*. An agent is a piece of software that conforms to some formal *specification*. Each agent is executed on its own *platform*. The platform provides execution environment and interface to the location middleware. Agents communicate through the special construct of the coordination space called *scope*. An agent can cooperate only with agents participating in the same set of scopes. Agents can logically and physically migrate from a location to a location. Migration from a platform to a platform is also possible using logical mobility. An agent is built on the base of one or more *roles*. Role is a formal functionality specification and composition of specifications of all the roles forms the specification of the agent. A role is the result of the decomposition of an abstract *scope model* and a *run-time scope* is an instantiation of a abstract scope model.

A *location* is a container for scopes. It can be associated with a particular physical location and can have certain restrictions on the types of supported scopes. It provides means for communication and coordination between agents and hence constitutes the core part of the system. We assume that each location has a unique name in the given context. To automatically create new scopes and restrict access to existing ones, location keeps track of present agents and their properties.

A *platform* provides an execution environment for an agent. It is composed of a virtual machine for code execution, networking support and middleware for interaction with location. A platform may be supported by PDA, smart-phone, laptop or a location server. The concept of platform is important to clearly differentiate between a location providing coordination services to agents and middleware that only supports agent execution.

An *agent* is a piece of software implementing a set of roles. The roles allow the agent to take part in certain scopes. All agents must implement the minimal functionality called the default role, which specifies activities outside scopes.

A *scope* is a dynamic container for tuples. It provides an isolated coordination space for compatible agents. This is achieved by restricting visibility of the tuples contained in the scope to the participants of the scope. Scopes are initiated by an agent and then atomically created by location when all the participants are ready. Scopes can be nested and scope participants can create new contained scopes. Scope is defined by the set of roles and a set of logical restrictions.

A scope becomes *activated* after some agent creates it with the *CreateScope* operation. A scope is *open* when there are some vacant roles in it, and is *closed* when all the roles are taken. A scope is *pending* if some required roles are not taken yet and *expanding* if all the required roles are taken but there still some vacant roles. Closed and expanding states correspond to working scopes, where agents can communicate. All its participants of a pending scope are blocked until the scope state is changed into closed or expanding.

A *role* is an abstract description of the agentf functionality. Each role is associated with

some scope type. An agent may implement a number of roles and can also play several roles in the same scope or different scopes. The CAMA approach emphasises the context-awareness of mobile agents. The context of an agent in CAMA systems consists of the following parts: a set of locations the agent is connected to, the state of scopes in which the agent is currently participating (including tuples contained in these scopes) and role attributes of other agents collaborating with the agent.

6.3 Formal Development Process

The formal development process of the CAMA system consists of several steps. First, we create abstract specifications of the middleware (location) and the scopes that will be supported by the system. Then we develop (by the stepwise refinement method) specifications of different roles participating in scopes. Finally, we compose an agent specification as a combination of several developed roles (i.e., agent interfaces) and the default functionality defining the agent behaviour outside scopes. The agent specification can be further refined by adding more details and custom functionality. The compatibility of different agents is ensured by the fact that all agents were developed by the formal refinement method from the same abstract specifications of different roles and the middleware. Therefore, agents can collaborate making safe assumptions about the functionality of their peers. The development process is conducted within the formal framework of the B Method [Abr96] (further referred to as B), which is an approach to the industrial development of highly dependable software. The tool support available for B provides us with the assistance for the entire development process.

6.3.1 Development of scopes and roles

The specification of a scope describes general functionality of several collaborating agents (in particular roles). The task of formal development is to use the specification as the starting point for the derivation of specifications of the corresponding agent roles (interfaces). To guarantee correctness of the resulting role specifications, we use formal refinement and decomposition techniques. On the other hand, we have to take into account scope nesting, when scopes have embedded subscopes providing an extended functionality. Subscope specifications can be naturally derived from the original scope specification via refinement. After verifying the correctness of refinement, we can continue the development process by decomposing the specification into corresponding roles as described above.

As a result, we have two orthogonal development processes with the same starting point - the original specification of a scope. Both developments arrive at role specifications describing agent functionality in the corresponding scopes. However, the hierarchy of scopes and subscopes should be reflected in the corresponding specifications of agent roles. Hence the roles in subscopes must be the extensions of the corresponding roles in the scopes. In other words, to guarantee the consistency of developed roles, we have to show that the subscope roles refine the corresponding scope roles.

6.3.2 Agent Design

Agent design starts with selection of roles that the agent must implement. It can implement any number of roles from different scopes. Initially roles inside of an agent are totally independent specifications that may well correspond to several independent processes running in an agent. Agent refinement specifies additional operations that control agent behaviour during migration, location selection, scope creation and joining, and other activities not covered by roles. During agent refinement process, the agent roles can also be refined, possibly by adding some new functionality. Due to the nature of refinement, the refined roles are still compatible with the original abstract roles.

We start building an agent specification by extending one or more roles obtained formally through the decomposition of abstract scope models. The refinement step introduces a specification of the minimal agent functionality called the default role. This functionally permits an agent to talk to locations, create/join/leave scopes, and migrate. The agent may also need some logic that glues independent interfaces and allows them to talk to each other. This is done via the global agent variables and the special methods for accessing them. After the agent specification is ready, it is used to build the source code for the actual agent program. The standard work cycle of an agent looks like this: an agent detects the available locations and connects to at least one of them, then looks for current activities on the location(s) or creates its own new scope, and finally joins a scope and plays one of the implemented roles in it. Only when the agent decides to play a particular role in a scope, it really starts to cooperate with other agents. The agent is capable of understanding its peers since the role functionalities of all the scope participants are based on the same abstract model. As a result, the composition of agent functionalities in a scope corresponds to the initial abstract model. The correctness of a model instantiation, which means that the scope instantiates the corresponding abstract scope model, can be demonstrated by analysing the agent design process and assuming that there is a correct transition from agent model to agent implementation.

6.3.3 B specification of the middleware

To ensure correct behaviour of the location-based system, the middleware of the location should enforce a certain discipline on agents. For instance, the properties of the scopes defined upon scope creation are preserved in spite of volatile connectivity and dynamic nature of scopes. Moreover, this should guarantee the integrity of information about agents in locations and scopes. These complex interdependencies should be stated explicitly and verified. We have developed a formal specification of the location middleware which is the core of the system [ILRT05]. It corresponds to the most complex part of the system and not only defines the operations that the location provides to support communication between agents but also states the properties of data structures in the location. The actual middleware implementation will be based upon this formal model.

6.4 Discussion

In this chapter we presented our advances in developing the methodology (based on formal methods) that will allow us to fully model and build the mobile location-based systems. While

designing our methodology we have been influenced by the requirements document written for the Ambient Campus case study [A+05]. During the first year we have focused on modelling middleware that supports our mobile agent abstractions. The most significant achievement is the formal B specification of the location, i.e. the core part of the middleware presented in [ILRT05]. The selection of the location-based architecture (discussed in [IR05]) has influenced all the parts of our work on the case study, including the methodology. As a future work, we are planning to investigate more closely the agent design process and apply it to the full design of the lecture scenario of the Ambient Campus case study [A+05]. We will also conduct several extensive experiments covering the full cycle of system development - starting from an abstract system model and making all the steps until we get a running piece of software.

Chapter 7

Methodology for Formal Model-Driven Development of Communicating Systems

7.1 Introduction

Modern telecommunicating systems are usually distributed software-intensive systems providing a large variety of services to their users. Nokia Research Centre has developed a design method *Lyra*[LTO04] – a UML-based[RJB98] service-oriented method specific to the domain of communicating systems and communication protocols. The design flow of Lyra is based on concepts of decomposition and preservation of the externally observable behaviour. The system behaviour is modularised and organized into hierarchical layers according to the external communication and related interfaces. It allows the designers to derive the distributed network architecture from the functional system requirements via a number of model transformations. This approach coincides with the stepwise refinement paradigm adopted in the B Method. We propose a set of formal specification and refinement patterns reflecting the essential models and transformations of Lyra. Our approach is based on stepwise refinement of a formal system model in the B Method.

While formalizing Lyra, we single out a generic concept of a communicating service component and propose patterns for specifying and refining it. In the refinement process the service component is decomposed into a set of service components of smaller granularity specified according to the proposed pattern. Moreover, we demonstrate that the process of distributing service components between different network elements can also be captured by the notion of refinement. The proposed formal specification and development patterns establish a background for automatic generation of formal specifications from UML models and expressing model transformations as refinement steps. Via automation of the UML-based Lyra design flow we aim at smooth incorporation of formal methods into existing development practice.

7.2 Pattern for specifying service component in B

Modelling Service Component in B. The service-oriented development approach is based on decomposition of system behaviour. The notion of a service provides a convenient mechanism for modelling and reasoning about system interactions. Therefore, the service-oriented development paradigm is particularly suitable for the development of reactive systems, i.e., systems

running in constant interaction with their environments.In the service-oriented development a service component as a coherent piece of functionality which provides its services to a service consumer via Provided Service Access Point – PSAP. The notion of a service component is a convenient abstraction to represent service providers at the different levels of abstraction. Indeed, on a high level of abstraction even the entire communicating system can be seen as a service component. On the other hand, peer proxies introduced at the lowest level of abstraction can also be seen as the service components providing the physical data transfer services. Therefore, the notion of a service component is central to the entire Lyra development process.

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

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

In our specification we abstract away from the details of computations required to execute the service. Our specification of ACAM is merely a statement non-deterministically generating results of service execution in case of success or failure. The communication with a service component is conducted via two channels – inp_chan and out_chan – shared between the service component and the service consumer. While specifying a service component, we adopt a systems approach, i.e., model the service component together with the relevant part of its environment, the service consumer. Namely, we model how the service consumer places requests to execute a service in the operation env_req and reads the results of service execution in the operation env_resp.

The operations read and write are internal to the service component. The service component reads the requests to execute a service from inp_chan as defined in the operation read. As a result of read execution, the request is stored into the internal data buffer input, so it can be used by ACAM while performing the required computing. Symmetrically the operation write models placing the results of computations performed by ACAM into the output channel, so it can be read by the service consumer. We reserve the abstract constant NIL to model the absence of data, i.e., the empty channels. The operations discussed above model the ACM part of ACC.

We argue that the machine ACC can be seen as a specification pattern which can be instantiated by supplying the details specific to a service component under construction. For instance, the ACM part of ACC models data transfer to and from the service component very abstractly. While developing a realistic service component, this part can be instantiated with real data structures and corresponding protocols for transferring them. In the next section we demonstrate how Lyra development flow can be formalized as refinement and decomposition of ACC.

```
MACHINE  ACC
VARIABLES  inp_chan, input, out_chan, output
INVARIANT
```

```
  inp_chan : INPUT_DATA & input : INPUT_DATA &
 out_chan : OUT_DATA &   output : OUT_DATA

INITIALISATION
  inp_chan, input := INPUT_NIL, INPUT_NIL ||
  out_chan, output := OUT_NIL, OUT_NIL

OPERATIONS

/* ACM */
env_req =
  SELECT inp_chan = INPUT_NIL
  THEN
    inp_chan :: INPUT_DATA - {INPUT_NIL}
  END;

read =
  SELECT not(inp_chan = INPUT_NIL) & (input = INPUT_NIL)
  THEN
    input,inp_chan := inp_chan,INPUT_NIL
  END;

write =
  SELECT not(output = OUT_NIL) & (out_chan = OUT_NIL)
  THEN
    out_chan,output := output,OUT_NIL
  END;

env_read =
  SELECT not(out_chan = OUT_NIL)
  THEN
    out_chan := OUT_NIL
  END

/* ACAM */

calculate =
  SELECT not(input = INPUT_NIL) & (output = OUT_NIL)
  THEN
    CHOICE
      output :: OUT_DATA - {OUT_NIL,OUT_FAIL}
    OR
      output := OUT_FAIL
    END ||
    input := INPUT_NIL
  END;
```

```
END
```

7.3 Formal Service-Oriented Development

In Lyra, a service component is usually represented as an active class with the PSAP attached to it via the port. The state diagram depicts signalling scenario on PSAP including the signals from and to the external class modelling the service consumer. Essentially these diagrams suffice to specify the service component according to the pattern ACC proposed in the previous section.

The UML2 description of PSAP of the service component is translated into the ACM part of the corresponding machine. The ACAM part of this machine instantiates the non-deterministic assignment of ACC by the data types specific to the modelled service component. These translations formalize the Service Specification phase of Lyra.

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

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

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

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

Assume that the service component *SC* specified by the machine *ACC_SC* at the Service Specification phase is providing the service *S* which is decomposed into the subservices *S*1, *S*2, and *S*3. Moreover, let assume that the state machine of Service Director defines the desired order of execution: first *S*1, then *S*2 and finally *S*3. In B such decomposition can be represented as a refinement of our abstract pattern ACC instantiated to model SC as shown below:

```
REFINEMENT ACC_R1_SC

REFINES ACC_SC

VARIABLES
  curr_service, handling_flag

INVARIANT
  curr_service : {SD, S1, S2,S3} &
  handling_flag : BOOL & ...

INITIALISATION
  curr_service, handling_flag := SD,FALSE || ...

OPERATIONS

/* ACM  */
...

/* ACAM' */

S1 = SELECT curr_service = S1
          THEN  handling_flag := TRUE
          END;
S2 = ...
S3 = ...

director =
  SELECT handling_flag = TRUE
  THEN
    IF curr_service = SD
    THEN
       curr_service := S1
    ELSIF curr_service = S1
    THEN
       S1_data :: S1_DATA-{S1_NIL};
        curr_service := S2
    ELSIF curr_service = S2 ...
    ELSIF curr_service = S3 ...
    END ||
    handling_flag := FALSE
  END;

calculate =
  SELECT (curr_service=CALC) & ...
  THEN
```

```
    output,input := OUT_data,INPUT_NIL ||
    curr_service := SD
  END;
END
```

This step focuses on refinement of the ACAM part of ACC. As in ACAM, in the refinement of it - ACAM'- the operation calculate puts the results of service execution on the output channel. However, calculate is now preceded by the operation director, which models Service Director orchestrating the stages of execution. We introduce the variables S1_data, S2_data and S3_data to model the results of execution of the corresponding stages. The operation director specifies the desired execution flow by assigning corresponding values to the variable curr_service. In general, execution of any stage of service can fail. In its turn, this might lead to failure of the entire service provision. Here we omit the presentation of failures of service provision and error recovery while specifying Service Director. The detailed description of this can be found in [LTL+05].

Unlike in Lyra, in our B development the Service Decomposition and Service Distribution phases are not entirely disjoint. This is explained by the fact that the INCLUDES structuring mechanism enforces the master-slave relationship between components, i.e., the including machine has complete control over the included machine. As a result, modelling of communication between two peer components is cumbersome. However, this problem can be alleviated if the targeted service distribution is taken into account while introducing the communication with the external service components via USAPs.

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

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

In the first case, the service component *SC* plays a role of the service consumer for the service components *SC*1, *SC*2 and *SC*3. We specify the service components *SC*1, *SC*2 and *SC*3 as separate machines *ACC_SC*1, *ACC_SC*2, *ACC_SC*3 according to the proposed pattern ACC. The process of translating their UML2 models into B is similar to specifying *SC* at the Service Specification phase. The ACM parts of the included machines specify their PSAPs. To ensure the match between the corresponding USAPs of *SC* and PSAPs of the external service components, we derive USAPs of *SC* from PSAPs of *SC*1, *SC*2 and *SC*3.

Besides defining separate machines to model external service components, in this refinement step we also define the mechanisms for communicating with them. We refine the operation director to specify communication on USAPs. Namely, we replace non-deterministic assignments modelling stages of service execution by the corresponding signalling scenario: at the proper point of the execution flow director requests a desired service by writing into the input channel of the corresponding included machine, e.g., SC1_write_ichan, and later reads the produced results from the output channel of this machine, e.g., SC1_read_ochan.

```
REFINEMENT ACC_R2_SC

REFINES ACC_R1_SC

INCLUDES
    ACC_SC1, ACC_SC2, ACC_SC3

/* ACM of ACC_SC */
...

/* ACAM'' */

director =
  SELECT handling_flag = TRUE
  THEN
    IF curr_service = SD
    THEN
       curr_service := S1
    ELSIF curr_service = S1
    THEN
       SC1_write_ichan(input);
       S1_data <- SC1_read_ochan
    ELSIF curr_service = S2 ...
    ELSIF curr_service = S3 ...
    END ||
    handling_flag := FALSE
  END;

calculate =...

END



MACHINE ACC_SC1
...

/* ACM  of ACC_SC1 */
SC1_write_ichan(SC1inp) ...

SC1read ...

SC1out<- - SC1_read_ochan ...

SC1write ...
```

```
/* ACAM of ACC_SC1 */
calculate...

END
```

Modelling the case of the distributed service director is more complex. Let assume that the execution flow of the service component SC is orchestrated by two service directors: the $ServiceDirector1$, which handles the communication on PSAP of SC and communicates with $SC1$, and $ServiceDirector2$, which orchestrates the execution of $S2$ and $S3$.

The service execution proceeds according to the following scenario: via PSAP of SC $ServiceDirector1$ receives the request to provide the service S. Upon this, via USAP of SC, it requests the component $SC1$ to provide the service $S2$. After the result of $S2$ is obtained, $ServiceDirector1$ requests $ServiceDirector2$ to execute the rest of the service and return the result back. In its turn, $ServiceDirector2$ at first requests $SC2$ to provide the service $S2$ and then $SC3$ to provide service $S3$. Upon receiving the result from $S3$, it forwards it to $ServiceDirector1$. Finally, $ServiceDirector1$ returns to the service consumer the result of the entire service S via PSAP of SC.

This complex behaviour can be captured in a number of refinement steps. At first, we observe that $ServiceDirector2$ co-ordinating execution of $S2$ and $S3$ can be modelled as a "large" service component $SC2 - SC3$ which provides the services $S2$ and $S3$. Let us note that the execution flow in $SC2 - SC3$ is orchestrated by a "centralized" service director $ServiceDirector2$. We use this observation in our next refinement step. Namely we refine the B machine modelling SC by including into it the machines modelling the service components $SC1$ and $SC2 - SC3$ and introducing the required communicating mechanisms. In our consequent refinement step we focus on decomposition of $SC2 - SC3$. The decomposition is performed according to the proposed scheme: we introduce the specification of $ServiceDirector2$ and decompose ACAM of $SC2 - SC3$. Finally, we single out separate service components $SC2$ and $SC3$ as before and refine $ServiceDirector2$ to model communication with them. We omit the presentation of the detailed formal specifications – they are again obtained by the recursive application of the proposed specification and refinement patterns.

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

Chapter 8

Exception Handling in Coordination-based Mobile Environments

Mobile agent systems have many attractive features including asynchrony, openness, dynamicity and anonymity, which makes them indispensable in designing complex modern applications that involve moving devices, human participants and software. To be comprehensive this list should include fault tolerance, yet as our analysis shows, this property is, unfortunately, often overlooked by middleware designers. A few existing solutions for fault tolerant mobile agents are developed mainly for tolerating hardware faults without providing any general support for application-specific recovery. In our recent paper [IR05] we introduce a novel exception handling model that allows application-specific recovery in coordination-based systems consisting of mobile agents. The essence of this model is in supporting flexible exception propagation from one mobile communicating agent to another (or to a set of chosen agents) to ensure correct recovery in the situations when the agent cannot handle exceptions internally.

In this work we are focusing on coordination-based mobile environments, which are becoming the dominating paradigm in developing many mobile applications. In such systems agents communicate through a tuple space using Linda-type coordination primitives allowing them to put tuples in a tuple space shared by several agents, get them out and test for them.

8.1 Exception Propagation through Coordination Space

Our mechanism of the exception propagation is complimentary to the application-level exception handling. All the recovery actions are implemented by application-specific handlers. The ultimate task of the mechanism is to transfer exceptions between agents in reliable and secure way. However the enormous freedom of behaviour in agent-based systems prevents from delivering guarantees of reliable exception propagation in a general case. Although we can clearly identify the situations when exceptions may be lost or not delivered within a predictable time period. If application requires cooperative exception at certain moments then for that moments, agents behaviour must be constrained to prevent any unexpected migrations or disconnections.

For agents there are three basic operation available to receive and send inter-agent exceptions. They are supplementary to the application-level mechanism and their functionality do not overlap.

throw	**wait**	**check**

The first operation, `throw`, propagates an exception to an agent or a scope. Important requirements is that the sending agent prior to sending an exception must have got a message from the destination agent and they both must be in the same scope. These two variants of the operation has the following form:

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

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

Also the mechanism itself does not introduce any limitations on agent activities. Though agents dynamicity an reliability of exception propagation are conflicting concepts we believe that it is developers who must take the final decision to favour either of them. Notion of openness is the key for building large-scale agent systems. Proper exception handling was proved to be crucial for proper component composition. It is even more so for mobile agent where composition is dynamic and parts of the system are developed independently. To support large-scale compositions of exceptions-enhanced agents we are going to elaborate a formal step-wise development procedure.

Two other operations, `check` and `wait` are used to explicitly poll and wait for inter-agent exceptions.

- `check` - if there are any exception pending for the calling agents raises exception `E(e)` which is a local envelop for the oldest pending exception.
- `wait` - waits until any inter-agent exception appears for the agent and raises it in the same way as the previous operation.

8.2 Traps Mechanism

The propagation procedure expressed only with the primitives above would be too limited and inflexible for mobile agent systems. To control propagation process in a way that account for various agent behaviour scenario we introduce a notion of *trap*. Trap is a set of rule created by agents that controls exception propagation and exists independently of the creating agent. Location providing a coordination space is storing and manipulating traps. A trap is essentially a list of rules chosen depending upon the currently propagate exception.

A trap can be enabled when there is an incoming message for the agent that created the trap. Agent may have several traps and traps are automatically organized into an hierarchical structure. When an exception appears, the most recently added trap is activated. If the trap fails to find a handling rule for the exception, the exception is propagated to the second most-recent trap and so on. Agents can dynamically create, add and remove traps. The following operations are used to build traps:

- `deliver` - delivers the thrown exception to the destination agent. The exception is stored until the destination agent is ready to react to it or the containing scope is destroyed;
- `relay(t)` - propagates the exception to trap `t` which may be a trap of another agent. Name of a trap can be only learned through negotiations with the trap owner. Owner of the trap becomes the destination the propagated exception;
- `abort` - leaves the current trap and transfers control to the next in the hierarchy.
- `if` *(condition)* `then` *ac* - action *ac* is applied conditionally;
- . *(concatenation)* - forms a new action by concatenation of two other actions.

The `deliver` operation was designed to tolerate agent migration and connectivity fluctuations. It introduces some level of asynchrony and makes the whole exception propagation scheme more suitable to the asynchronous communication style of coordination space. `relay` operation is the tool for building linked trap structures for disciplined cooperative exception handling. Discussion and examples on this can be found in [IR05].

Preconditions for the `if` operation are formed from the following primitives:

- `local` - holds if the owner of the trap is currently connected to the hosting location
- `local(a)` - holds if agent `a` is currently connected to the hosting location
- `tuple(t)` - holds if there is a tuple matching template `t`
- $\neg$, $\vee$, $\wedge$ - logical operations that can be used on the predicates above

Rule preconditions and concatenation form a very expressive mechanism that may form traps for many interesting scenarios. For example a rule in trap could make multiple delivers to involve several agents, or, depending upon the locality of the trap owner, an exception may be routed to the agent itself, another agent or even a trap in a different location.

8.2.1 Discussion

Our model meets all major requirements for exception handling in mobile coordination systems that we set when we started our work. It supports asynchronous raising and handling of exceptions, ensures agent anonymity, imposes no restrictions on the agent behavior or its internal activity, supports migrating agents and works effectively for both loosely- and tightly-coupled communication patterns.

We have applied the general ideas behind the mechanism in the context of the Lime middleware [PMR99]. More specifically we have developed an extension of the Lime system and conducted a number of experiments to check the implementation. The full code of our implementation can be downloaded from *http://www.cs.ncl.ac.uk/ alexander.romanovsky/home.formal/limeh.zip*. The complete description of the API and Lime extended operations can be found in [IR05].

Chapter 9

The way ahead

This "Month 12" report obviously only indicates out initial steps on "Rodin Methods". Progress has already been made on some important issues and we remain convinced that "tensioning" our evolving ideas on methodology against the chosen case studies will yield a valuable contribution to formal approaches to fault tolerance.

9.1 Link to Rodin Tasks

The Rodin Description of work defines the following tasks, the link with the sections above is as follows

Task	Description	**C**hapter or **S**ection
T2.1	Formal representations of architectural design, decomposition and mapping principles	S2.1.1, S2.2, C4, C6, C7
T2.2	Reusability, genericity, refinement	S2.5, C4, C5, C6, C7
T2.3	Development templates for fault-tolerant design methods	S2.4, C3, C6, C7, C8
T2.4	Development templates for reconfigurability, adaptability and mobility	C6, C7, C8
T2.5	Requirements evolution and traceability	S2.1, S2.5, C3, C6

9.2 Addressing the issues in the next period

It would be rash to suggest that we will resolve all of the methodological issues identified above even by the end of the whole Rodin project (even foolhardy to promise this by the "Month 24" report): some of the questions noted above have been tackled by the world's leading researchers for more than a decade.

But we feel that it is important to consider even long-standing questions as we evolve and test the Rodin methodology and tools. We might find resolutions; we can at least (try to) avoid compounding old problems with what look like independent decisions.

We do undertake to report on our evolving thoughts on each of these issues in future deliverables from this work package (rather than just describe a single method and omit to mention any limitations).

The next report should be able to report worked examples from the case studies.

9.3 Relevant publications not cited above

Readers might be interested in [WV01, Jon81, dRE99, Owi75, OG76, Jon83a, Jon83b, Jon96a, Stø90, Col94, Xu92, Bue00, Din00, BS01, dR01, Plo81b, Plo03b, Plo03a, Jon96a, Bur04, Nip04, CM92].

Bibliography

[A$^+$05] Budi Arief et al. Rodin deliverable d4 - traceable requirements document for case studies. Technical report, Project IST-511159, February 2005.

[Abr85] J.R. Abrial. Programming as a mathematical exercise. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 113–139. Prentice-Hall International, 1985.

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr04] J.-R. Abrial. Mechanical press: Requirement document. Private communication, Nov. 2004.

[ACD$^+$03] Tony Andrews, Franciso Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, version 1.1. http://www.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[AE04] Schneider S. A. and Treharne H. E. Verifying controlled components. In *Proceedings of IFM '04, Integrated Formal Methods*, LNCS 2999. Springer, 2004.

[ALLR04] A. Avizienis, J.-C. Laprie, C. Landwehr, and B. Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–13, 2004.

[AM02] J.R. Abrial and L. Mussat. On using conditional definitions in formal theories. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B. pages 242-269*. Springer-Verlag, 2002. LNCS 2272.

[BBF$^+$05] Roberto Bruni, Michael Butler, Carla Ferreira, Tony Hoare, Hernan Melgratti, and Ugo Montanari. Reconciling two approaches to compensable flow composition. In *CONCUR 2005*, 2005.

[BBN$^+$03] L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In *Global Computing*, volume LNCS 2874 of *LNCS*, pages 88–150. Springer-Verlag, 2003.

[BCJ84] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BDK01] E. Best, R. Devillers, and M. Koutny. *Petri Net Algebra. EATCS Monographs on TCS*. Springer, 2001.

[BF00] Michael Butler and Carla Ferreira. A process compensation language. In *Integrated Formal Methods(IFM'2000)*, volume 1945 of *LNCS*, pages 61 – 76. Springer-Verlag, 2000.

[BF04] Michael Butler and Carla Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Coordination 2004*, volume 2949 of *LNCS*. Springer-Verlag, 2004.

[BFN05] Michael Butler, Carla Ferreira, and M.Y. Ng. Precise modelling of compensating business transactions and its application to BPEL. *Journal of Universal Computer Science*, 11(5), 2005.

[BHF04] Michael Butler, Tony Hoare, and Carla Ferreira. A trace semactics for long-running transaction. In A.E. Abdallah, C.B. Jones, and J.E. Sanders, editors, *Proceedings of 25 Years of CSP*, volume 3525 of *Springer LNCS*, London, 2004.

[BJ05] J. I. Burton and C. B. Jones. Investigating atomicity and observability. *Journal of Universal Computer Science*, 11(5):661–686, 2005.

[BJRT05] M. Butler, C. B .Jones, A. Romanovsky, and E. Troubitsyna. Proceedings of the workshop on rigorous engineering of fault-tolerant systems (reft 2005). Technical Report CS-TR-915, ISSN 1368-1060, School of Computing Science, University of Newcastle, April 2005.

[BL05] Michael Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In John Fitzgearld, Ian Hayes, and Andrzej Tarlecki, editors, *Proceedings of Formal Methods 2005, Newcastle*, LNCS 3582, pages 221–236. Springer, 2005.

[BMM05] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL*, pages 209–220, 2005.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.

[Bue00] Martin Buechi. *Safe Language Mechanisms for Modularization and Concurrency*. PhD thesis, Turku, 2000.

[Bur04] J. Burton. *The Theory and Practice of Refinement-After-Hiding*. PhD thesis, University of Newcastle upon Tyne, 2004.

[But00] Michael Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Asp. Comput.*, 12(3):182–198, 2000.

[But02] Michael Butler. A System-based Approach to the Formal Development of Embedded Controllers for a Railway. *Design Automation for Embedded Systems.*, 6:355–366, 2002.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A systematic Introduction*. Springer Verlag, 1998.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[CM92] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY*. PhD thesis, Louvain-la-Neuve, June 1994.

[Col04] Joseph W Coleman. Features of BPEL modelled via structural operational semantics. MPhil thesis, University of Newcastle Upon Tyne, November 2004.

[Col05] Joey W. Coleman. Examining BPEL's compensation construct. Technical Report Series CS-TR-894, University of Newcastle Upon Tyne, March 2005.

[CSW03] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for Circus. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.

[Dan02] J. Daniels. Modeling with a sense of purpose. *IEEE Software*, pages 8–10, January/February 2002.

[Dij82] Edsger W Dijkstra. On making solutions more and more fine-grained. In *Selected Writings on Computing: A Personal Perspective*, pages 292–307. Springer-Verlag, 1982. (Orignially EWD622, written 26 May 1977).

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[DKK04] R. Devillers, H. Klaudel, and M. Koutny. Petri net semantics of the finite pi-calculus. In *FORTE*, volume LNCS 3235 of *LNCS*, pages 309–325. Springer-Verlag, 2004.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[dRE99] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

[Eva04] Eric Evans. *Domain Driven Design*. Addison Wesley, 2004.

[Fau00] S.R. Faulk. Product-line requirements specification (PRS): an approach and case study. In *Proc. Fifth IEEE International Symposium on Requirements Engineering*. IEEE Comput. Soc, Aug. 2000.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.

[FJ90] J.S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 189–210. Springer-Verlag, 1990.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling systems: practical tools and techniques in software development*. Cambridge University Press, 1998.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2005.

[Fuc92] N. E. Fuchs. Specifications are (preferably) executable. *IEE, Software Engineering Journal*, 6:323–334, Sepember 1992.

[GB96] B. Grahlmann and E. Best. Pep - more than a petri net tool. In *TACAS*, volume LNCS 1055 of *LNCS*, pages 397–401. Springer-Verlag, 1996.

[Gel85] D. Gelernter. Generative communication in linda. *ACM Computing Surveys*, 7(1):80–112, 1985.

[GH96] Andy Gravell and Peter Henderson. Executing formal specification need not be harmful. *IEE/BCS Software Engineering Journal*, March 1996.

[GIJ+02] M.-C. Gaudel, V. Issarny, C. B. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R.J. Stroud, and F. Taini. Final version of DSoS conceptual model. Technical Report CS-TR-782, School of Computing Science, Newcastle University, 2002.

[Gre89] T. R. G. Green. Cognitive dimensions of notations. *People and Computers*, 5, 1989.

[Hal96] Anthony Hall. Using formal methods to develop an atc information system. *IEEE Software*, 13(2), 1996.

[Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice Hall International, second edition, 1993.

[HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE, Software Engineering Journal*, 4(6):320–338, November 1989.

[HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.

[HJN94] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. *ACM Software Engineering News*, 19(3):75–81, July 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[ILRT05] A. Iliasov, L. Laibinis, A. Romanovsky, and E. Troubitsyna. Towards formal development of mobile location-based systems. In *REFT'05 – Workshop on Rigorous Engineering of Fault Tolerant Systems*, July 2005.

[IR05] A. Iliasov and A. Romanovsky. Exception handling in coordination-based mobile environments. In *29th Annual International Computer Software and Applications Conference Edinburgh, Scotland*, pages 341–350. IEEE CS Press, July 2005.

[ISO95] ISO. VDM-SL. Technical Report Draft International Standard, ISO/IEC JTC1/SC22/WG19 N-20, 1995.

[Jac96] B. Jacobs. Inheritance and cofree constructions. In *Proceedings of the European Conference on Object-Oriented Programming*, volume LNCS 1098. Springer, 1996.

[Jac00] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2000.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomicity manifesto: a story in four quarks. *ACM SIGMOD Record*, 34(1):63–69, 2005.

[JM94] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980. ISBN 0-13-821884-6.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[Jon95] C.B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

[Jon96a] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon96b] C. B. Jones. TANSTAAFL (with partial functions). In William Farmer, Manfred Kerber, and Michael Kohlhase, editors, *Proceedings of the CADE-13 Workshop on the Mechanization Of Partial Functions*, pages 53–64, 1996.

[Jon03] C. B. Jones. Wanted: a compositional approach to concurrency. In A. McIver and C. Morgan, editors, *Programming Methodology*, pages 1–15. Springer Verlag, 2003.

[Jon05] C. B. Jones. Sequencing operations and creating objects. In *Proceedings Tenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 33–36. IEEE Computer Society, 2005.

[Kho03] V. Khomenko. Model checking based on prefixes of petri net unfoldings. phd thesis. Technical report, School of Computing Science, University of Newcastle upon Tyne, 2003.

[Lam97] W. Lam. Achieving requirements reuse: a domain-specific approach from avionics. *Journal of Systems and Software*, 38(3):197–209, Sept. 1997.

[LB03] M. Leuschel and M. Butler. ProB: a model checker for B. volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Verlag, 2003.

[LTL+05] L. Laibinis, E. Troubitsyna, S. Leppanen, J. Lilius, and Q. Malik. Formal model-driven development of communicating systems. Technical Report 691, TUCS, 2005.

[LTO04] S. Leppänen, M. Turunen, and I. Oliver. Application driven methodology for development of communicating systems. In *Proc. of* FDL'04 – Forum on Specification and Design Languages, 2004.

[MD98] Brendan P. Mahony and Song Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, 1998.

[MH91] B Mahony and I Hayes. Using continuous real functions to model timed histories. In P. Bailes, editor, *Engineering Safe Software*, pages 257–270. Australian Computer Society, 1991.

[Mid93] Cornelius A. Middelburg. *Logic and Specification: Extending VDM-SL for advanced formal specification*. Chapman and Hall, 1993.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[MS01] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Sci. Comput. Program.*, 40(1):59–96, 2001.

[NFP98] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[Nip04] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a java-like language. Manuscript, Munich, 2004.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Old02] Paul Oldfield. Domain modelling. Technical report, Technical Report of Appropriate Process Movement, http://www.aptprocess.com, 2002.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975. 75-251.

[Plo81a] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, University of Aarhus, 1981.

[Plo81b] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Plo03a] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Functional and Logic Programming*, 2003. forthcoming.

[Plo03b] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Functional and Logic Programming*, 2003. forthcoming.

[PMR99] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. *Proc of the 21st Int. Conference on Software Engineering (ICSE'99), Los Angeles (USA)*, 1999.

[Rei85] W. Reisig. *Petri Nets. An Introduction. EATCS Monographs in Computer Science*. Springer, 1985.

[RFW+04] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1998.

[SB04] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[SBEJ04a] C. Snook, M. Butler, A. Edmunds, and I. Johnson. Rigorous development of reusable, domain-specific components, for complex applications. In J. Jurgens and R. France, editors, *Proc. 3rd International Workshop on Critical Systems Development with UML*, pages 115–129, Lisbon, 2004.

[SBEJ04b] C. Snook, M. Butler, A. Edmunds, and I. Johnson. Rigourous development of reusable domain-specific components for complex applications. In J. Jurgens and R. France, editors, *Proc. 3rd Intl. Workshop on Critical Systems Development with UML*, Lisbon, 2004.

[SM92] Sally Shlaer and Stephen Mellor. *Object-oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, 1992.

[Smi97] Graeme Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *Proceedings FME '97*, LNCS 1313, pages 62–81. Springer, 1997.

[SOB04a] C. Snook, I. Oliver, and M. Butler. The UML-B profile for formal systems modelling in UML. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems*, chapter 5. Springer, 2004.

[SOB04b] C. Snook, I. Oliver, and M. Butler. The UML-B profile for formal systems modelling in UML. In *UML-B Specification for Proven Embedded Systems Design*. Springer, 2004.

[SPJ05a] C. Snook, M. Poppleton, and I. Johnson. The engineering of generic requirements for failure management. In *Proc. 11th International Workshop on Requirements Engineering: Foundation for Software Quality (in press)*, Porto, June 2005. Essener Informatik Beitrage.

[SPJ05b] C. Snook, M. Poppleton, and I. Johnson. Towards a methodology for rigorous development of generic requirements patterns. In *Proc. REFT'05 (in press)*, Newcastle, UK, July 2005.

[ST04] Steve Schneider and Helen Treharne. Verifying controlled components. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings Integrated Formal Methods, IFM 2004, Canterbury, UK*, LNCS 2999, pages 87–107. Springer, 2004.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. Available as UMCS-91-1-1.

[Val98] A. Valmari. The state explosion problem. In *Advances in Petri Nets*, volume LNCS 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.

[WC02] J.C.P. Woodcock and A. Cavalcanti. The semantics of Circus. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *Proceedings ZB 2002, Grenoble, France*, LNCS 2272, pages 184–203. Springer, 2002.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2001.

[XRR+99] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of a safety-critical system based on coordinated atomic actions. In *Proc. of 29th Int. Symp. Fault-Tolerant Computing*. IEEE Computer Society Press, 1999.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.