



Project DEPLOY
Grant Agreement 214158

*“Industrial deployment of advanced system engineering methods
for high productivity and dependability”*



DEPLOY Deliverable D44

**D6.6 Deploy Methods:
Final Report**

Public Document

April 27, 2012

<http://www.deploy-project.eu>

Contributors:

J.R. Abrial, D. Basin, J. Bendisposto, J. Bryans, M. Butler,
M. Daneshtalab, I. Dinca, A. Edmunds, J. Fitzgerald,
A. Fuerst, A. Gondal, S. Hallerstede, T. Hoang, A. Iliasov,
M. Jackson, M. Jastram, C. Jones, M. Kamali, V. Kozyura,
L. Ladenberger L. Laibinis, M. Leuschel, I. Lopatkin, I. Maamria,
M. Mazzara, L. Petre, D. Plagge, M. Poppleton, Y. Prokhorova,
A. Rezazadeh, A. Roth, M. Sarshogh, M. Schmalz, K. Sere,
R. Silva, A. Stefanescu, A. Tarasyuk, E. Troubitsyna,
L. Tsiopoulos, W. Wei, S. Wieczorek, S. Yeganeh

Editors:

M. Butler, A. Edmunds, A. Rezazadeh,

Reviewers:

Timo Latvala, Elena Troubitsyna

Contents

I	Executive Summary	6
1	Executive Summary	7
1.1	Introduction	7
1.2	The Event-B Language	8
1.3	From Requirements to Models	9
1.4	Model Animation	11
1.5	Dependability	12
1.6	Composition and Reuse	14
1.7	Modelling Hardware Architectures	16
1.8	Shared Variable Concurrency	17
1.9	Code Generation	18
1.10	Timing and Probabilities	19
1.11	Event-B and Choreography Models	20
1.12	Structured Data and Mathematical Extensions	21
1.13	Model-Based Test Generation	22
II	Detailed Contributions	24
2	Event-B Overview	25
2.1	The Event-B Language	25
2.2	Incremental construction of an example	27
2.3	The Event-B Proof Method	32
3	From Requirements to Models	39
3.1	The Problem Frames Approach	39
3.2	Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation	58
3.3	Requirements Traceability	75

4	Model Animation	96
4.1	Refinement Animation	96
5	Dependability	117
5.1	Modelling dependability requirements	117
5.2	Describing exceptional behaviour (i.e. fault modelling)	121
5.3	Fault-tolerance in Design	128
5.4	Derivation and Formal Verification of a Mode Logic for Layered Control Systems	135
5.5	Security	150
6	Composition and Reuse	152
6.1	Decomposition	152
6.2	Modules	153
6.3	Reuse via Application of Modelling and Refinement Patterns .	157
6.4	A Survey on Event-B Decomposition	182
6.5	Formal Feature-Oriented Reuse in Event-B	214
7	Modelling Hardware Architectures	241
7.1	Refinement-Based Modelling of 3D NoCs	241
7.2	Model-Based Analysis Tools for Component Synthesis	254
8	Concurrency	271
8.1	Concurrency	271
8.2	Rely/guarantee and Event-B	290
9	Code Generation	292
9.1	Code Generation with Tasking Event-B	292
9.2	Code Generation and Scheduling of Event-B Models	327
10	Timing and Probabilities	344
10.1	Modelling Timing Properties In Event-B	345
10.2	Towards Probabilistic Modelling in Event-B	369
10.3	Quantitative Verification of Safety	381
10.4	Continuous behaviour	395
10.5	Probabilities	399
11	Event-B and Choreography Models	401
11.1	Introduction	401
11.2	Checking Consistency Between Message Choreographies And Their Implementation Models	402

12 Structured Data and Mathematical Extensions	418
12.1 Structured Data-Types and Refinement	418
12.2 Term Rewriting in Logics of Partial Functions	431
12.3 Theory Plug-in and Mathematical Extensions	436
13 Model-Based Test Generation	453
13.1 Introduction	453
13.2 MBT using explicit model-checking	455
13.3 MBT using constraint solving	461
13.4 MBT using model learning	471
13.5 Test suite reduction	499
13.6 Conclusion	510
Bibliography	510

Part I

Executive Summary

Chapter 1

Executive Summary

M. Butler, A. Edmunds

1.1 Introduction

Methodological work in the DEPLOY Project was primarily carried out in three workpackages:

- WP6 which is concerned with requirements
- WP7 which is concerned with composition and reuse
- WP8 which is concerned with dependability, especially fault-tolerance

Previous DEPLOY work on methods was reported in D15 and D31. D15, produced at M17, reported on methods covering requirements, reuse and dependability. This was largely work-in-progress that set the scene for later work. As the industrial deployments progressed during Years 1 and 2 several challenges, arising from deployment of the formal methods and tools, (Event-B and Rodin) were identified and addressed. D31, produced at M32, described several avenues of methodological research that addressed these challenges.

This deliverable (D44) reports on methods developed and used in the DEPLOY Project, primarily during the final 2 years of DEPLOY. During this period, the methods matured as they were applied to the deployment case studies as well as to other related test benches. The deliverable is intended to serve two purposes: firstly to provide an executive overview of the main achievements on methods in DEPLOY; and secondly, to provide a detailed

technical resource that can be used by researchers and practitioners in formal methods beyond the lifetime of DEPLOY. The executive overview is provided in this chapter while the technical resource is provided by the remaining chapters. The detailed technical resource represents a synthesis of research results developed by DEPLOY partners including theoretical developments, methodological developments and guidelines on the practical application of the research results.

Much of the deployment of formal methods in DEPLOY was based on the Event-B language and associated Rodin toolset. During DEPLOY these were extended and complemented by a range of techniques and methods. This deliverable provides general technical background on the Event-B method including structuring and proof of models, which is a pre-requisite for understanding many of the later contributions. It then covers a range of methods that extend or complement Event-B and Rodin, providing motivations, insights, technical definitions, and usage guidelines illustrated by case studies.

There is a strong interplay between methods and deployments as well as methods and tools. While this deliverable touches on both deployments and tools, the focus is on the principles and general guidelines. Challenges and assessment related to specific industrial deployments are addressed in D43 (Final Assessment and Integration Results) as well as in the DEPLOY Book. Technical details on tooling achievements are provided in D45 (Model construction tools and analysis tools IV).

The sections of this chapter follow the structure of the remaining chapters so Section 1.2 summarises the contribution in Chapter 2 and so on.

1.2 The Event-B Language

Chapter 2 provides the general technical background for Event-B. It essentially represents a small subset of the comprehensive exposition of Event-B in Abrial’s book [Abr10]. The technical background is included here in order to make the later material more accessible.

Event-B is defined in terms of a few simple concepts that describe a discrete event system, and the proof obligations that permit verification of properties of the event system. An Event-B *model* consists of *contexts* and *machines*. Contexts contain the static parts of a model. These are *constants* and *axioms* that describe the properties of these constants. Machines contain the dynamic parts of a model. A machine is made of a *state*, which is defined by means of *variables*. Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions, numbers, etc. They are constrained by *invariants* $I(v)$ where v are the variables of the ma-

chine. Invariants are supposed to hold whenever variable values change, but this must be proved first. Besides its state, a machine contains a number of *events* which specify how the state may evolve. Each event is composed of a *guard* and an *action*. The guard is the necessary condition under which the event may occur. The action, as its name indicates, determines the way in which the state variables are going to evolve when the event occurs. An event may have *parameters* that are local to that event. Parameters can serve different functions, for instance, to model arrays of events or as communication channels in composition of machines [But09b]. An event may be executed only when its guard holds. Events are *atomic* and when the guards of several events hold simultaneously, then *at most one of them* may be executed at any one moment. The choice of event to be executed is non-deterministic.

While the high-level structuring of a model uses some simple concepts (contexts, machines, guards, actions), the richness of Event-B is provided by the use of set theory as the language for expressing values (abstract and concrete) and properties of values. The core Rodin tool comes with a rich implementation of set theory, built-in. One of the major achievements in DEPLOY was the provision of mechanisms to extend the mathematical theories supported by Rodin and this is covered in Chapter 12.

The Event-B chapter covers the standard proof obligations associated with Event-B models including well-definedness, invariant preservation and refinement. Machine refinement provides a means to introduce more detail about the dynamic properties of a model. The theory of refinement is a simplified form of the corresponding notion of the Action Systems formalism [Bac90] that has inspired the development of Event-B. Action Systems and other refinement theories support both forward and backward refinement. In common with the B-Method, Event-B refinement currently supports forward refinement; backwards refinement is not currently supported.

The Event-B chapter also outlines the language features that are designed to assist with proof, namely witness and theorems. Witnesses help the automated discharge of existential obligations associated with refinement while theorems are forms of assertions that follow from axioms, invariants and guards and help guide proofs.

1.3 From Requirements to Models

It comes as no surprise that the quality of requirements documentation has a major influence on the construction and validation of formal models and this was borne out strongly by the industrial deployments in DEPLOY notably the automotive deployments by Bosch. A common problem in many

industrial requirements documents is the mixing of problem description and solution description; many requirements documents are more like design documents than high-level descriptions of functional requirements. Chapter 3 covers approaches to requirements elucidation, structuring, tracing and using the requirements structure to guide the construction of formal models. The chapter covers Jackson’s problem frames approach that was used extensively by Bosch in their deployment, the so-called MCC (monitored, controlled, commanded) guidelines for operator driven control systems and traceability between requirements and Event-B models. The intention of the approaches described in these chapter is to focus attention more on the problem than the solution in high-level requirements analysis and to structure requirements in such a way that facilitates traceability with the formal models.

1.3.1 The Problem Frames Approach

The focus of the DEPLOY project is on the development of software for computer-based systems. In developing such a system, the central—though not the only—goal is to ensure satisfaction of the functional requirements in the physical world in which the computing equipment is embedded. We briefly illustrate the Problem Frames approach, and explain its rationale. The approach is not presented in detail here, but the discussion should be understandable even if the approach is unfamiliar. A basic account is available in a book [Jac01], and some further developments of the approach and its underlying principles are discussed in more recent papers [Hal07, BGJ06, Jac10, JHJ07].

1.3.2 Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation

Good requirements structure can greatly facilitate the construction of formal models of systems. This section describes an approach to requirements structuring for control systems that aims to facilitate refinement-based formalisation. In addition to the well-known monitored (M) and controlled (C) phenomena used to analyse control systems, we also identify commanded (C) phenomenon, reflecting the special role that an operator plays in system control. These MCC system phenomena guide the structure of the requirements analysis and documentation as well as the structure of the formal models.

We model systems using the Event-B formalism, making use of refinement to support layering of requirements. The structuring provided by the system

phenomena and by the refinement layers supports clear traceability and validation between requirements and formal models. As a worked example, we structured the requirements of an automotive lane departure warning system using this approach. We found missing requirements through this process and we evolved the requirement document through domain experts’ feedback and formal modelling.

We discuss the evolution of the requirement document of an automotive case study (LDWS - Lane Departure Warning System) through domain expert’s feedback and modelling using Event-B. The MCC modelling guidelines [But09c] inspired us to structure the requirement document based on monitored, controlled and commanded phenomena. In addition, some criteria for layering the requirements and mapping informal requirements to a formal model were provided.

1.3.3 Requirements Traceability

In this section, we describe an incremental approach to requirements validation and systems modelling. Formal modelling facilitates a high degree of automation: it serves for validation and traceability. The foundation for our approach is requirements that are structured according to the WRSPM reference model. We provide a system for traceability with a state-based formal method that supports refinement. We do not require all specification elements to be modelled formally and support incremental incorporation of new specification elements into the formal model. Refinement is used to deal with larger amounts of requirements in a structured way. The use of the ProR plug-in for Rodin to support this approach to requirements traceability is also described.

1.4 Model Animation

The Rodin tool automatically generates proof obligations that can be analysed to gain insight into a formal model. They often give good indications of how to make an improvement in case of inconsistencies in a model. However, there are also many occasions where proof obligations do not point directly to a problem or where a model does not contain inconsistencies but is still “incorrect” (see, e.g., the Earley parser example discussed in [BLLS08]). In many such cases animation is a useful tool to gain further insight into a model.

ProB was originally written for the B method and had a very rudimentary support for animating refinements. In this chapter we describe how we

improved ProB's animation of refinements. ProB helps users locate the cause of problems in a model; and points to specific invariants, guards, and so on, that have unexpected values.

We describe a method for animation and validation of Event-B models. The key ingredient that makes the algorithm tractable are the witnesses of Event-B. We have implemented the algorithm within ProB, and have shown how a variety of refinement errors can now be detected effectively. We have applied the technique to various case studies, and have animated up to 14 levels simultaneously. The algorithm used by ProB delivers improved performance when compared to previous versions, for instance, by applying symmetry reduction to multiple levels of refinement at once.

Animation does not need to provide all of the capabilities that formal proof provides, because we consider proof and animation complementary techniques of validation. For instance, for finding candidates for invariants, formal proof appears superior; for checking whether they are always satisfied, i.e., finding counter examples, animation appears superior. Moreover, animation can be used to reason about properties that have not (yet) been formalised.

1.5 Dependability

In the DEPLOY project, we sought to use formal methods to enhance the dependability of systems, in particular the types of systems developed by industrial partners. The pilot studies provided a base and a "reality check" to this work. In working with the industrial partners, an incompatibility we observed was in the languages and terms that we used. Classical dependability research is presented in terms of faults, errors and failures; but the language preferred by industrial partners (to describe the same phenomena) used terms such as exception or incompatibility.

In the DEPLOY project, we sought to use formal methods to enhance the dependability of systems, in particular the types of systems developed by industrial partners. The deployments provided a base and a "reality check" to this work. In working with the industrial partners, an incompatibility we observed was in the languages and terms that we used. Classical dependability research is presented in terms of faults, errors and failures, but the language preferred by industrial partners (to describe the same phenomena) was terms such as exception or incompatibility. The subject however was clearly of interest to the industrial partners - an analysis of the resilience requirements of each of the pilot studies was carried out, which found that many of the requirements were related to fault-tolerance features.

In Chapter 5 we provide general guidelines on modelling dependability requirements, on modelling exceptional/faulty behaviour and modelling fault tolerance mechanisms using flows. We also describe the use of a fault analysis technique (FMEA) together with representations of modes in Event-B. The chapter concludes with a brief overview of experience with modelling access control using Event-B.

1.5.1 Dependability Requirements

The objective of this section is reporting the main ideas which have been brought forward and developed on dependability requirements in the automotive domain by Newcastle and Bosch . A digested understanding of the issues investigated over the duration of DEPLOY is presented in an attempt to offer a broad view of the research after the close and fruitful collaboration. We outline an approach of elaborating and decomposing problem descriptions and then linking these to Event-B models.

1.5.2 Describing Exceptional Behaviour

In this section we bring together the work describing exceptional behaviour (or “faults”), that we carried out within DEPLOY. The analysis of the resilience requirements of each of the pilot studies is outlined. Work with SAP is presented, which considered a scenario where a choice had to be made between two sets of infrastructure guarantees in the implementation of a protocol design. We were able to demonstrate that one of the sets of guarantees was not strong enough to support the protocol as designed. We also discuss the work on describing modal systems for modelling fault tolerance in Event-B.

1.5.3 Fault-tolerance in Design

This Section presents two modelling environments developed for Event-B, and shows their relevance to fault-tolerant design. It begins with a discussion of the *Flow* modelling environment, and contains a description of a tool designed to model the flow of control through an Event-B model. The second part (on fault tolerance views) is again a description of a modelling environment, this time designed to aid in the construction of fault tolerance features in a concise manner and to formally link them to Event-B models.

1.5.4 Derivation and Formal Verification of a Mode Logic for Layered Control Systems

While designing layered mode-rich systems, we should ensure mode consistency and guarantee that the mode logic also caters to fault tolerance. In this section we propose to conduct Failure Modes and Effects Analysis (FMEA) of each operational mode to identify mode transitions required to implement fault tolerance. The general idea of the development process presented in this section is to start from an abstract specification of the upper architectural layer and gradually unfold lower layers by refinement. Since refinement allows us to develop a system in a correct-by-construction fashion, stepwise unfolding of the architectural layers also guarantees preserving mode consistency between lower and upper layers.

1.5.5 Security

Security is an important aspect of overall system dependability. Although security was not a major feature of the industrial deployments in the project, there was a degree of interest in understanding how the topic could be addressed through a refinement-based methods. In [HBA09], we proposed an approach for developing access control system using Event-B. The key aspect of our approach is to develop independently the insecure target system and the security authorisation. The resulting systems are then combined together (e.g., using *Parallel composition* [Sil]).

1.6 Composition and Reuse

Chapter 6 outlines several techniques that support composition and reuse in Event-B modelling. All these techniques are supported by Rodin plug-ins developed during DEPLOY.

1.6.1 Decomposition

In this section we provide useful insights and guidelines for model decomposition, drawn from our experiences of realistic software modeling in collaboration with industrial partners. These guidelines are not supposed to provide a single solution for all possible decompositions. Instead, we describe various stages of decomposition and point out which design problems need to be addressed at each stage. We highlight design problems that can be easily overlooked. Finally, we compare three different decomposition approaches,

and discuss their strengths and weaknesses when applied to different kinds of software systems.

1.6.2 Modules

Recently the Event-B language and tool support have been extended with a capability to define modules [ITL⁺10b, Modb] – components containing groups of “callable” operations. Modules can have their own (external and internal) state and invariant properties. The important characteristic of modules is that they can be developed separately and then composed with the main system.

1.6.3 Reuse via Application of Modelling and Refinement Patterns

This section presents three different approaches for facilitating reuse by application of pre-defined solutions – design and refinement patterns. The developed patterns are integrated into the refinement process of Event-B. We proposed a theoretical basis for automation of the refinement process. We introduced the notion of refinement patterns – model transformers that generically represent typical refinement steps. Refinement patterns allow us to replace a process of devising a refined model and discharging proof obligations by a process of pattern instantiation. While instantiating refinement patterns, we reuse not only models but also proofs. All together, this establishes a basis for automation. In this section we also demonstrate how to define refinement patterns for the Event-B formalism and described a prototype tool allowing us to automate refinement steps in Event-B.

1.6.4 A Survey on Event-B Decomposition

The primary challenge of a decomposition refinement step is to ensure that the structure of a model, before decomposition, fits the requirements of the chosen decomposition style; and that this leads to helpful sub-models that can be developed separately, with a tangible advantage in proof effort and overall model scale. As with any top-down approach for system development using refinement, the more abstract models that we have initially, the more useful our decomposition step will be. In this work we define guidelines for decomposition techniques and draw some comparison between the models, developed using different decomposition techniques.

1.6.5 Formal Feature-Oriented Reuse in Event-B

A software product line (SPL) is a set of related products which share a common base; and have sufficient flexibility to meet variable user requirements [CN01]. We have developed a framework that combines the strengths of both formal methods and SPL engineering, inspired by the work of Snook et al. [SPJ08]. This means that if we build a database of Event-B models for a PL that have already been proved, we can specify various products of the PL by configuring and composing these models in different ways. We have developed a feature modelling tool (inspired by earlier feature modelling tools, e.g., [AC04]) as a plug-in to Rodin. The tool can be used to draw feature models, configure them, and resolve any conflicts that arise during composition. Our case-study examples showed that we can utilise existing Event-B (de)composition techniques to decompose a system into various features, refine them and recompose later to model the desired PL instance. We present two case-studies that model different systems, in different styles, to explore reusability; and, at the same time, explore Event-B's potential for feature-modelling. These also highlighted further tooling requirements and research questions. We also proposed some guidelines that we think would be useful for SPL modelling, using Event-B's current tools and techniques.

1.7 Modelling Hardware Architectures

Chapter 7 covers approaches to using Event-B for modelling and reasoning about hardware designs in Event-B. Although not directly related to the industrial deployments in the DEPLOY project they are based on collaboration with electronic design engineers and they build on the composition techniques covered in Chapter 6.

1.7.1 Refinement-Based Modelling of 3D NoCs

This section proposes three increasingly detailed formal models for a 3D NoC together with the constraints for proving correctness. We illustrate the formal modelling of the XYZ routing algorithm as a case study and discuss the associated proof obligations.

1.7.2 Model-Based Analysis Tools for Component Synthesis

In this work, we propose a formal method based methodology for component synthesis. We understand the functionality of a component as a collection of *services* that the component has to implement [Szy98]. Components have been traditionally developed with an emphasis on the specification of their functionality, without explicitly describing and analysing their interdependencies. Here, based on the services that components need to implement, we identify some initial boundaries of components. Then, we identify those services needed for communication among components; hence, we identify the component interdependencies. Based on these interdependencies, we then reason about the suitability of certain boundaries, i.e., we argue whether some components should or not merge. As an application of our approach, we employ these interdependencies to place components on hardware platforms, so that highly communicating components find each other in their vicinity.

1.8 Shared Variable Concurrency

In Chapter 8 we consider programs that use several co-operating parallel processes in order to compute the intended final result. Proving correctness of such programs is a difficult task because of the interleaved execution of many sub-statements from different processes. These sub-statements may be executed in an unpredictable order. As a result, techniques such as program testing do not give us sufficient confidence about the correctness of these programs, since no execution leading to an error might appear during tests. To achieve correctness, it is therefore necessary to develop these programs and prove them formally.

There are a number of methods for proving the correctness of parallel programs [dRdBH⁺01]. Chapter 8 describes an approach applying the technique of refinement and decomposition in Event-B [Abr09] to modelling and reasoning about concurrent algorithms. The approach defines a four stage process starting with a simple abstraction, followed by a refinement stage, then a decomposition stage and then further rerfinement.

Chapter 8 also includes a broader reflection on the rely/guarantee approach and its relationship to the Event-B approach as outlined in the chapter.

1.9 Code Generation

Code generation is an important part of the formal engineering tool chain, which enables complete support for development from high-level models down to executable implementations. We address the need for specification of, and code generation for, sequential and concurrent implementations. Our work in DEPLOY has a particular emphasis on real-time embedded software. Chapter 9 covers two main approaches to code generation from Event-B models. In the first approach, Event-B is augmented with sequential control structures that guide the structure of the generated code. In effect the modeller is defining the scheduling of events through a collection of concurrent sequential programs. In the second approach, the scheduling of events is done in an interleaved manner over a several processors so that the modeller does not provide explicit scheduling.

1.9.1 Code Generation with Tasking Event-B

Previously we proposed an approach for code generation from Event-B, tailored to Java implementations of concurrent programs. The experience gained in that undertaking has informed our new approach. In particular, our interest has now turned towards the multi-tasking software used with embedded control systems. We wish to generate Ada and C code, since these languages are often used to implement these systems. We have developed a new methodology, and extended the existing Event-B tool to facilitate code generation. The approach has been formulated so that it integrates well with the existing Event-B methodology and tools. Event-B is extended with AutoTask, Shared, and Environ machines. We use these features to specify how elements should be translated to code, i.e. controller tasks, protected objects and environment tasks. Using refinement, decomposition, and the extension, we structure projects so that they are amenable to automatic code generation. This necessitates the introduction of a great deal of low-level detail into the specification; so models grow in size considerably. To ensure models remain tractable, we decompose the models (perhaps a number of times). Decomposition encourages modular reasoning, and makes proof more tractable. AutoTask machines contain a behavioural description; making use of imperative, programming-like constructs. We also specify a task's priority and its life-cycle, i.e. whether it is periodic, triggered, repeating or one-shot. However, timing aspects are not modelled formally. The interaction between controller tasks and the environment, and controller tasks and protected objects, is described by the synchronised events that arise during model decomposition. These events can have one of several roles. They may

participate in a procedure call to a protected object, or they may take part in a sensing, or actuating, role - interacting with the environment. In the generated code, sensing and actuating may be simulated, using a type of subroutine call; or additional information can be provided to allow a task to read/write directly from/to a specified memory location. To validate the approach we undertook a case-study, which we use in the section as a running example.

1.9.2 Code Generation and Scheduling of Event-B Models

In this section, we suggest an approach to code generation in Event-B and scheduling of the resulting code. We have written a plug-in for the Rodin platform that translates models into C++ [Str00] code, which can then be compiled into object code. The generated code consists of C++ methods, which can be executed in parallel using a separate scheduler that we have developed. Scheduling depends on a behavioural semantics, which for our tool is inherited from the Action Systems formalism [BKS83], and which allows for a parallel interpretation of programs. We support parallel execution of independent events by using the MPI (Message Passing Interface) framework [Mes], which allows events to be distributed over several cores or processors, as well as over a network.

1.10 Timing and Probabilities

Clearly, timing is important in many embedded systems applications. While Event-B does not have explicit support for timing, it is possible to model discrete timing properties by introducing a discrete clock into models. Chapter 10 presents an approach to modelling discrete timing by using standard patterns of timing properties and their refinement. Three groups of discrete timing properties are introduced: deadlines, delays and expiries. Then our approach to formulate the process of encoding them in Event-B is discussed in detail. Our approach uses annotations to express timing properties; and, these are translated to invariants, guards and actions. Finally, since refinement is one of the most important features of Event-B, some patterns to refine abstract timing properties to concrete ones, based on control flow refinement, are introduced.

Probabilistic reasoning is important for analysing the reliability of safety-critical systems and Chapter 10 also describes approaches to extending Event-B to treat probability. A first approach considers cyclic systems and shows

that their behaviour can be represented via a common Event-B modelling pattern. We show then how to augment such models with probabilities (using a proposed probabilistic choice operator) that in turn would allow us to assess their reliability.

Related to this, Chapter 10 describes design strategies that allow the developers to structure safety requirements according to the system abstraction layers. Essentially, such an approach can be seen as a process of extracting a fault tree – a logical representation of a hazardous situation in terms of the primitives used at different layers of abstraction. Eventually, we arrive at the representation of a hazardous situation in terms of failures of basic system components. After augmenting Event-B specifications with an explicit representation of probabilities of component failures [TTL10], we can use standard calculations to obtain a probabilistic evaluation of a hazard occurrence. As a result, we obtain an algebraic representation of probability of safety requirements violation. This probability is defined using the probabilities of system component failures. To illustrate our approach, we present a formal development and safety analysis of a radio-based railway crossing.

1.11 Event-B and Choreography Models

Applying the concepts of Service-Oriented Architectures (SOA) has already become a mainstream in industry [Hef09]. The development of business applications according to these principles implies a layered design and implementation. Chapter 11 describes an industrial approach to the verification of a consistency relation between such layers. In our case service choreographies defined by Message Choreography Models (MCM) and their corresponding implementation models represented as Business Objects are examined. By translating both into Event-B specifications we are able to prove the consistency relation between them. A number of case studies with realistic industrial software models were carried out which showed the a of our verification technique. Apart from giving details about our concrete realization, we also discuss the general challenges that have to be faced when developing a verification approach applicable for the real-world systems.

The work focuses on a consistency problem: the one between message choreographies (more precisely, local partner models) and their implementation models. Implementation models are not final implementation code. Even though very close to actual implementation details, they specify only the aspects relevant to the changes of internal life cycles of business objects in terms of state transition graphs. Therefore, our work is *not* concerned with source code analysis. Like our previous approach in [KRW09], we check con-

sistencies through a translation into Event-B, a formal specification language supported by the Rodin platform [Abr10]. This work has been carried out in the context of SAP software developments, and we report on the experience we gained from it.

1.12 Structured Data and Mathematical Extensions

The mathematical language of Event-B is set theory and Rodin supports a rich set of set theoretic operators and related theories. However, it was recognised early on in DEPLOY that, using theories, we could provide a mechanism for extending the mathematical language; and provide a means for explicit representation of structured data, such as records. Chapter 12 describes the extension of the mathematical language, which has been realised through Rodin plug-ins.

1.12.1 Structuring Data: Structured Data-Types and Formal Modelling

As a part of formal modelling we may need to model structured data. A structured data-type can have multiple fields, or elements, which can be introduced at ones or gradually during successive refinement steps. The core mathematical language of Event-B currently does not support a syntax for the direct definition of structured types such as records or class structures. Nevertheless, it is possible to model structured types using standard constructs provided in Event-B. Structured data-types then can be extended/enriched with extra fields in accordance with the general refinement approach. In this section we show how this can be achieved.

1.12.2 Term Rewriting in Logics of Partial Functions

We summarize the contributions of the DEPLOY project to term rewriting in logics of partial functions. The main contribution is *directed rewriting*, a technique that avoids solving well-definedness conditions during term rewriting. Directed rewriting already appears in ML, an automated theorem prover for classical B, and PVS [ORS92], although we are not aware of a written documentation or justification. Butler and Maamria [MB10b] develop a theoretical foundation for a restricted version of directed rewriting. Schmalz [Sch11b] shows how to overcome these limitations and provides a positive

report on its practical impact. Although the technique of directed rewriting is described in the context of Event-B, it has applications in other logics of partial functions such as LPF [JM94], the logic underlying VDM, and PVS. See [Sch11b] for detailed explanations.

1.12.3 Theory Plug-in and Mathematical Extensions

We provide an overview of the work carried out to enhance the extensibility of Event-B toolset. In particular, we outline the approach that is adopted to enable the users of Rodin to contribute to the Event-B mathematical language and the proof infrastructure. Our aim is to improve the overall extensibility of Event-B to enhance usability and effectiveness of the methodology. We are primarily concerned with facilitating the addition of new operators (i.e., *language extensions*) and new proof rules (i.e., *prover extensions*) to suit end-users needs. It is essential to ensure that any technique that achieves these goals also has to maintain *practicality of use*, and ensure *soundness preservation*. Practicality of use is important to relieve end-users from writing Java code. Soundness preservation ensures that any extensions do not compromise the logical foundations of the formalism.

1.13 Model-Based Test Generation

In Chapter 13, we describe methods for generating test cases from Event-B models, including their prototype implementation. For model-based testing (MBT) using state-based models, test generation algorithms usually traverse the state space starting in an initial state trying to reach a certain coverage criteria (e.g. state coverage) collecting the execution paths in a test suite. Event-B models do not have an explicit state space, but its state space is given by value of the variables and the state is changed by the execution of events that are enabled in that state. The ProB tool performs well when analysing the state space of an Event-B model, being able to explore it, visualise it, and verify various properties using model checking algorithms. Such model checking algorithms can be used to explore the state space of Event-B models using certain coverage criteria (e.g., event coverage) and thus, can generate test cases during the traversal. Moreover, the input data that triggers the different events provides the test data associated with the test cases.

1.13.1 Explicit Model-checking

We show how model-checking algorithms are implemented and applied to message choreography models from SAP. They work fine for models with data with a small finite range. However, in case of variables with a large range (e.g. integers), the state space explosion problem creates difficulties; this is because the model-checker explores the state by enumerating the many possible values of the variables.

1.13.2 Data Abstraction and Constraint Solving

To avoid the state space explosion due to the large bounds of the variables, we ignore these values in the first step and use the model-checker only to generate abstract test cases satisfying the coverage criteria. However, these paths may be infeasible in the concrete model due to the data constraints along the path. The solution is to represent the intermediate states of the path as existentially quantified variables. The whole path is then represented as a single predicate consisting of the guards and before-after predicates of its events. ProB's improved constraint solver is then used to validate the path feasibility and find appropriate data satisfying the constraints.

1.13.3 Model Learning

Event-B models are essentially abstract state machines. However, their states are not given explicitly; instead, they can be implicitly derived from the values of the model variables. Since the notion of state is at the heart of MBT, we provide a model-learning approach that uses the notion of cover automata to iteratively construct a subset of a state space together with an associated test suite. The iterative nature of the algorithm fits well with the notion of refinement from the Event-B method.

1.13.4 Test suite reduction

Large test suites generated by automatic test generators usually need to be optimised according to different criteria. We follow a more general approach of defining the reduction criteria as multi-objective test suite optimisation problems. They are solved using two modern Multi-Objective Evolutionary Algorithms, namely: NSGA-II [DAPM00] and SPEA-2 [ZLT01]. The experiments have been conducted using five test suites generated from two industrially-inspired Event-B models.

Part II

Detailed Contributions

Chapter 2

Event-B Overview

M. Butler, S. Hallerstede

2.1 The Event-B Language

Event-B is defined in terms of a few simple concepts that describe a discrete event system and proof obligations that permit verification of properties of the event system. The syntax of Event-B is not fixed in order to allow for easy extension of the language. However, we present the notation using some syntactical conventions. The keywords **when**, **then**, **end**, and so on, are just delimiters to make the textual representation more readable.

An Event-B *model* consists of *contexts* and *machines*. In this description we focus on machines. A fuller description of Event-B can be found in [Abr10].

Contexts contain the static parts of a model. These are *constants* and *axioms* that describe the properties of these constants.

Machines contain the dynamic parts of a model. A machine is made of a *state*, which is defined by means of *variables*. Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions, numbers, etc. They are constrained by *invariants* $I(v)$ where v are the variables of the machine. Invariants are supposed to hold whenever variable values change. But this must be proved first (see Section 2.3.2).

Besides its state, a machine contains a number of *events*, which specify how the state may evolve. Each event is composed of a *guard* and an *action*. The guard is the necessary condition under which the event may occur. The action, as its name indicates, determines the way in which the state variables

are going to evolve when the event occurs. An event may have *parameters* that are local to that event. Parameters can serve different functions, for instance, to model arrays of events or as communication channels in composition of machines [But96a].

An event may be executed only when its guard holds. Events are *atomic* and when the guards of several events hold simultaneously, then *at most one of them* may be executed at any one moment. The choice of event to be executed is non-deterministic. An event, named **evt**, is specified in one of the three following forms:

$$\begin{aligned} \text{evt} &\hat{=}\text{ any } t \text{ where } P(t, v) \text{ then } S(t, v) \text{ end} \\ \text{evt} &\hat{=}\text{ when } P(v) \text{ then } S(v) \text{ end} \\ \text{evt} &\hat{=}\text{ begin } S(v) \text{ end} \quad , \end{aligned}$$

where $P(\dots)$ is a predicate denoting the guard, t denotes parameters that are local to the event, and $S(\dots)$ denotes the action that updates some variables. The variables of the machine containing the event are denoted by v . The first event form is the most general one where an event has some parameters t and a guard $P(t, v)$. It can be executed in a state represented by v provided $P(t, v)$ holds for some t ; its effect on v is specified by action $S(t, v)$. The second event form is used if an event does not have any parameters. The third form is used if an event does not have parameters and its guard is true.

An action consists of a collection of *assignments* that modify the state simultaneously. An assignment has one of the following three forms:

Assignment	–	Before-After Predicate
$x := E(t, v)$	–	$x' = E(t, v)$
$x :\in E(t, v)$	–	$x' \in E(t, v)$
$x : Q(t, v, x')$	–	$Q(t, v, x') \quad ,$

where x are some variables, $E(\dots)$ denotes an expression, and $Q(\dots)$ a predicate. Assignments of the form $x := E(t, v)$ are called *deterministic*, the other two forms are called *nondeterministic*. Form $x :\in E(t, v)$ assigns x to an element of a set, and form $x :| Q(t, v, x')$ assigns to x a value satisfying a predicate. Simultaneity of a collection of assignments is expressed by conjoining the before-after predicates of an action. Variables y that do not appear on the left hand side of an assignment of action do not change. Formally this is achieved by conjoining $y' = y$ to the before-after predicate of the action.

In order to be able to provide better tool support, invariants, guards, actions are lists of named predicates and assignments. These names can be

used to refer to these objects from within the documentation of a machine. But foremost, these names are used to identify all objects and provide helpful information about the origin of proof obligations in the prover interface. The different predicates in the list are implicitly conjoined.

The mathematical language of Event-B has a simple type system. The types are *basic types* (such as integers or given sets that are specific to a model) or are formed from the *cartesian product* and *powerset* type constructors. Structures such as relations and functions are defined by combining these type constructors. A type inference system is used to infer types of constants, variables and event parameters from axioms, invariants and guards.

2.2 Incremental construction of an example

In this section we outline the construction of a small Event-B model and its refinement using the Rodin tool. Our aim is to illustrate the interaction between modelling and proof during model construction. This will serve to motivate the reactive nature of the support provided by the Rodin tool as models are constructed incrementally. Although we present this example before presenting the details of the proof method (Section 5), it is sufficiently self-explanatory for the general reader at this stage. The proof obligations being verified for the example are invariant preservation, refinement and well-definedness. We assume knowledge of basic set theory.

The model is of a system for checking registered users in and out of a building. We start the construction of the model by dealing only with registration of users. In the tool we create a new context and introduce a given set $USER$ in the context. We create a new machine and add a variable $register$ to the machine to represent the set of registered users. We create an invariant to specify the register as a set of users:

$$\text{inv1 } register \subseteq USER$$

The type inference mechanism infers the type of the $register$ variable, i.e., $\mathcal{P}(USER)$, from this invariant.

We create an event to add a new user to the register:

```

Register  $\hat{=}$  any  $u$  where
    grd1  $u \in USER \setminus register$ 
then
    act1  $register := register \cup \{u\}$ 
end

```

Here, parameter u represents the identity of the new user. At this stage in the formal development, we do not consider whether u is an input or an output of the system rather we simply state that the new user is not already in *register* (**grd1**).

With the above elements (set *USER*, variable *register*, invariant **inv1** and event *Register*) added to the project, the only error message we get is that the *register* variable has not been initialised. This is remedied by adding the action $register := \emptyset$ to the machine initialisation. At this stage the model results in no proof obligations since the invariant **inv1** is nothing stronger than a typing constraint.

Now we add variables to represent the set of people who are in the building (*in*) and those that are outside the building (*out*). These are typed and constrained to be subsets of *register* through the following invariants:

$$\begin{aligned} \text{inv2} \quad & in \subseteq register \\ \text{inv3} \quad & out \subseteq register \end{aligned}$$

Note that while these invariants allow the type inference mechanism to infer the types of *in* and *out*, they are stronger than typing invariants since *register* is a variable and not a type. We ensure that *in* and *out* are initialised to be empty. We have an obvious requirement that a user cannot be simultaneously inside and outside the building so we add a further invariant:

$$\text{inv4} \quad in \cap out = \emptyset$$

The resulting model now gives rise to 6 proof obligations in total; 3 of these are to verify that the initialisation establishes invariants **inv2** to **inv4** and 3 are to verify that the *register* event maintains invariants **inv2** to **inv4**. All 6 proof obligations are discharged automatically. The general definition of the proof obligations is explained in Section 2.3.

We add events to model users entering and leaving the building. Our first attempt at the *Enter* event is

```

Enter  $\hat{=}$  any  $u$  where
      grd1  $u \in out$ 
    then
      act1  $in := in \cup \{u\}$ 
    end

```

This event gives rise to 3 new proof obligations (1 for each of *inv2* to *inv4*), 1 of which is not automatically discharged. Using the proof obligation explorer

we can inspect this unproved proof obligation and see that it has hypotheses and a goal as follows:

$$\begin{array}{l}
\text{Hyp1 : } in \cap out = \emptyset \\
\text{Hyp2 : } u \in out \\
\vdash \\
\text{Goal : } (in \cup \{u\}) \cap out = \emptyset
\end{array} \tag{2.1}$$

Clearly this cannot be proved: if $u \in out$ then $\{u\} \cap out$ is not empty. Thus either the invariant it is associated with (*inv4*) is wrong or the *Enter* event is wrong and one or both need to be changed. The obligation explorer provides hyperlinks to both *inv4* and *Enter* to facilitate any changes to either. In this case we decide that the error is in the *Enter* operation since we neglected to remove the user from the variable *out*. We remedy this by clicking on the link to the *Enter* event and adding the following action to this event:

$$\text{act2 } out := out \setminus \{u\}$$

This addition results in all proof obligations being discharged automatically. Note that having a proof obligation that is not automatically discharged does not necessarily mean there is an error in the model. It may be due to a limitation of the automatic prover and instead the obligation may be provable using the interactive prover. The interactive prover of Rodin is explained in more detail in Section ??.

A further requirement on the model is that each registered user must either be inside or outside the building. Our existing invariants are not sufficient to express this property so we add a further invariant:

$$\text{inv5 } register \subseteq in \cup out$$

This addition gives rise to 3 new proof obligations, 1 of which is not automatically discharged:

$$\begin{array}{l}
\text{Hyp1 : } register \subseteq in \cup out \\
\text{Hyp2 : } u \in USER \setminus register \\
\vdash \\
\text{Goal : } (register \cup \{u\}) \subseteq in \cup out
\end{array} \tag{2.2}$$

Clearly this obligation is not provable: if u is not in *register*, then it is not in $in \cup out$. The obligation explorer tells us that this proof obligation arises from both *inv5* and the *Register* event. Inspection of the *Register* event shows

that it adds a user u to *register* but not to either *in* or *out*. We remedy this by deciding that newly registered users should be recorded as being outside the building and adding the following action to the existing *Register* event:

$$\text{act2} \quad \text{out} := \text{out} \cup \{u\}$$

All proof obligations of the resulting model are automatically discharged.

We now outline a data refinement of this model. Let us assume that we decide to implement this model as a simple database and replace the two abstract variables *in* and *out* with a single *status* function. The variables of the refined model (the concrete variables) are *register*, as before, and a new variable *status*, a total function from *register* to *STATUS* :

$$\text{inv6} \quad \text{status} \in \text{register} \rightarrow \text{STATUS}$$

STATUS is an enumerated type with distinct values *IN* and *OUT*.

The abstract *Enter* event is guarded by the condition that $u \in \text{out}$. In the refined *Enter* event, this guard is replaced by a condition on the *status* function. The refined event updates the *status* function rather than modifying the *in* and *out* variables:

$$\begin{aligned} \text{Enter} &\hat{=} \text{refines Enter} \\ &\text{any } u \text{ where} \\ &\quad \text{grd1} \quad u \in \text{register} \\ &\quad \text{grd2} \quad \text{status}(u) = \text{OUT} \\ &\text{then} \\ &\quad \text{act1} \quad \text{status}(u) := \text{IN} \\ &\text{end} \end{aligned}$$

The clause **refines Enter** indicates that the refined *Enter* event refines the *Enter* event of the abstract machine. In general, the names of refined events may differ from the corresponding abstract event so that a refined event must include an explicit reference to some event of the abstract machine. This refined event gives rise to an unproved refinement proof obligation as follows:

$$\begin{aligned} &\text{Hyp1 : } u \in \text{register} \\ &\text{Hyp2 : } \text{status}(u) = \text{OUT} \\ \vdash & \hspace{15em} (2.3) \\ &\text{Goal : } u \in \text{out} \end{aligned}$$

This proof obligation arises because of the need to show that the guard of a refined event implies the corresponding abstract guard. We can see that the hypotheses come from the guards of the refined event (**grd1** and **grd2**) while the goal is the guard of the abstract event. As it stands the goal cannot be proven since we have not stated any invariant relating the concrete *status* variable and the abstract *in* and *out* variables. Such an invariant is called a *gluing invariant* and will be described more precisely in the next section. Refinement proofs rely on such gluing invariants. We could simply convert the above proof obligation (2.3) into a gluing invariant as follows:

$$\begin{aligned} \text{inv7} \quad & \forall u \cdot u \in \text{register} \wedge \text{status}(u) = \text{OUT} \\ & \Rightarrow u \in \text{out} \end{aligned}$$

Intuitively this invariant is reasonable since it states that for any registered user whose status is *OUT* at the concrete level, that user is in the set *out* at the abstract level. Adding this invariant allows proof obligation (2.3) to be discharged automatically. A similar invariant about users in the set *in* can be used to discharge a proof obligation for a refined *Leave* operation. Note that invariant **inv7** refers to variables of both the abstract and refined machines. Invariants of a refined machine may refer to variables of the abstract machine. These are so-called *gluing invariants* and are explained further in Section 2.3.

We consider one other proof obligation associated with the refined *Enter* event:

$$\begin{aligned} & \text{Hyp1 : } u \in \text{register} \\ & \vdash \hspace{15em} (2.4) \\ & \text{Goal : } u \in \text{dom}(\text{status}) \wedge \\ & \hspace{10em} \text{status} \in \text{USER functional STATUS} \end{aligned}$$

This is a well-definedness obligation associated with the expression $\text{status}(u)$ in guard **grd2** of the refined *Enter* event. Function application in Event-B gives rise to a well-definedness obligation which requires that argument u is in the domain of *status* and that *status* is a *partial* function (and not just a relation). Because of the declaration of *status* (**inv6**), obligation (2.4) is discharged automatically. The well-definedness obligations are explained in more detail in Section ??.

We have now completed our construction of the small Event-B model and its refinement. With the old style tools for B, after constructing the model, we would have separately invoked the proof obligation generator and then the automatic prover. With the Rodin tool, this is taken care of automatically as we construct the model. Based on undertaking a range of Event-B

developments, large and small, with Rodin, our experience is that by making use of the feedback from the tool as we construct the model, e.g., the unproved proof obligations, we are guided towards construction of a model that has less errors and is more easily proved than if we were to delay any proof analysis until after constructing the full model.

2.3 The Event-B Proof Method

In this section we outline standard proof obligations associated with Event-B models.

2.3.1 Feasibility of Assignment

Recall from Section 2.1 that a nondeterministic assignment has the following form:

$$x := \mid Q(t, v, x').$$

Event-B requires actions to be feasible under the guard of the corresponding events, that is, when its guard is true the action of an event must yield a successor state. For the non-deterministic assignment we must prove

$$\begin{array}{c} I(v) \\ P(t, v) \\ \vdash \\ (\exists x' \cdot Q(t, v, x')) \end{array} ,$$

where $I(v)$ is the invariant of the machine and $P(t, v)$ the guard of the event.

2.3.2 Consistency of a Machine

Once a machine has been written, one must prove that it is *consistent*. This is done by proving that each event of the machine preserves the invariant. More precisely, it must be proved that the action associated with each event modifies the state variables in such a way that the modified variables satisfy the invariant, under the hypothesis that the invariant holds presently and the guard of the event is true. For a machine with state variable v , invariant $I(v)$, and an event **when** $P(v)$ **then** $v := E(v)$ **end** the statement to be

proved is the following:

$$\begin{array}{l}
I(v) \\
P(v) \\
\vdash \\
I(E(v)) \quad .
\end{array} \tag{2.5}$$

Note that, in practice we carry out a decomposition of (2.5) according to the lists of named invariants, guards, and actions. So statement (2.5) is not the proof obligation the user gets to see. Instead the user sees a collection of simpler proof obligations.

Inspection of (2.5) reveals the simplicity that is at the core of the Event-B method. In order to arrive at statement (2.5) we simply copy elements from the model and apply some basic rewriting. For the consistency proof obligation we copy the invariant $I(v)$ the guard $P(v)$ of the event in the hypothesis of the proof obligation and the modified invariant $I(E(v))$ where v has been replaced by $E(v)$ in the goal. This makes it easy to relate elements of the model to corresponding proof obligations when using Event-B in practice which is important when making incremental changes to a model as shown in Section 2.2. For example, proof obligation (2.1) is a consistency obligation that comes from `act1` of the abstract *Enter* event and `inv4`.

2.3.3 Refining a Machine

Machine refinement provides a means to introduce more detail about the dynamic properties of a model [AH07]. The theory of refinement is a simplified form of the corresponding notion of the Action Systems formalism [Bac90] that has inspired the development of Event-B. Action Systems and other refinement theories support both forward and backward refinement. In common with the B-Method, Event-B refinement currently supports forward refinement; backwards refinement is not currently supported.

Refining a machine consists of refining its state and its events. A concrete machine (with regards to the more abstract one) has a state that should be related to that of the abstraction by a so-called *gluing invariant*, which is expressed in terms of a predicate $J(v, w)$ connecting the abstract state represented by the variables v and the concrete state represented by the variables w . We introduce first refinement proof obligations for events without parameters to illustrate the principle. Afterwards, we show how we deal with parameters using *witnesses*. We deal with non-deterministic assignments similarly as explained in [Hal07].

Each event of the abstract machine is refined to one or more corresponding events of the concrete one. Informally speaking, a concrete event is said to refine its abstraction (1) when the guard of the former is stronger than that of the latter (guard strengthening), and (2) when the gluing invariant is preserved by the conjoined action of both events. In the case of an abstract event *abs* and a corresponding concrete event *con* of the form

$$\begin{aligned} \text{abs} &\hat{=} \text{when } P(v) \text{ then } v := E(v) \text{ end} \\ \text{con} &\hat{=} \text{when } Q(w) \text{ then } w := F(w) \text{ end} \quad , \end{aligned}$$

the statement to prove is the following:

$$\begin{aligned} &I(v) \\ &J(v, w) \\ &Q(w) \\ &\vdash \\ &P(v) \wedge J(E(v), F(w)) \quad , \end{aligned} \tag{2.6}$$

where $I(v)$ is the abstract invariant and $J(v, w)$ is the gluing invariant. Similarly to (2.5) the user never gets to see (2.6) but only the decomposed form.

In the case of events *abs* and *con* with parameters

$$\begin{array}{ll} \text{abs} \hat{=} \text{any} & \text{con} \hat{=} \text{any} \\ \quad t & \quad u \\ \text{where} & \text{with} \\ \quad P(t, v) & \quad t = W(u, w) \\ \text{then} & \text{where} \\ \quad v := E(t, v) & \quad Q(u, w) \\ \text{end} & \text{then} \\ & \quad w := F(u, w) \\ & \text{end} \quad , \end{array}$$

we have to prove:

$$\begin{aligned} &I(v) \\ &J(v, w) \\ &Q(u, w) \\ &\vdash \\ &P(W(u, w), v) \wedge J(E(W(u, w), v), F(u, w)) \quad , \end{aligned} \tag{2.7}$$

where $W(u, w)$ are called *witnesses*; see [Hal07]. Witnesses are specified in the model because they provide an essential insight into the refinement relationship of the abstract and the concrete event. (If a variable or parameter is repeated in a refinement, it is assumed that the concrete one and the abstract one are identical.) One could say, they provide a local gluing invariant. They also permit decomposition of (2.7) similarly to statement (2.5) in Section 2.3.2. Without the use of witnesses, the goal would be preceded by an existential quantifier $\exists t$ and the witness $W(u, w)$ would have to be provided interactively by the user during the proof of the obligation. By making the witness an explicit part of the model, its definition is more obvious to the modeller and refinement proofs go through more automatically.

2.3.4 Adding New Events in a Refinement

When refining a machine by another one, it is possible to *add new events*. Such events must be proved to refine a dummy event that does nothing (**skip**) in the abstraction. Moreover, it may be proved that the new events cannot collectively take control forever. For this, a unique *variant expression* $V(w)$ has to be provided, that is decreased by each new event. We refer to this as a *convergence* proof obligation.

In case the new event has the form:

$$\mathbf{evt} \triangleq \mathbf{when} \ R(w) \ \mathbf{then} \ w := G(w) \ \mathbf{end} \quad ,$$

the following statements (2.8) and (2.9) have to be proved:

$$\begin{array}{l} I(v) \\ J(v, w) \\ \vdash \\ J(v, G(w)) \end{array} \quad (2.8)$$

$$\begin{array}{l} I(v) \\ J(v, w) \\ \vdash \\ V(w) \in \mathcal{N} \wedge V(G(w)) < V(w) \end{array} \quad , \quad (2.9)$$

where we assume that the variant expression is a natural number (but it can be more elaborate).

2.3.5 Event extension

A very common form of refinement is called *superposition* refinement [Bac90] where only new elements are added to a machine and none of the existing variables or parameters is *data-refined* [Bv90]. In this case we only need to specify what is new in each refined event: new parameters, new guards, new actions. By doing this, making changes to a model becomes very efficient which is particularly important in conjunction with the incremental approach to modelling promoted by the Rodin tool.

In practice changes to a model occur as often to abstract machines as to refinements. Suppose, we have a model with 10 superposition refinements where we were to repeat the contents of all abstractions in each refinement. If we needed to change a guard of an event at the most abstract machine in that model it would be necessary to carry out the same change in all 9 refinements. The use of event extension means we only have to make a change in one place.

Let *abs* be an abstract event and *con* be an extension of *abs*:

$$\begin{aligned} \text{abs} &\hat{=} \text{ when } P(v) \text{ then } v := E(v) \text{ end} \\ \text{con} &\hat{=} \text{ when } Q(w) \text{ then } w := F(w) \text{ end} . \end{aligned}$$

Then the contents of event *abs* is automatically replicated in event *con*:

$$\begin{aligned} \text{con} &= \text{ when} \\ &\quad P(v) \wedge Q(w) \\ &\text{ then} \\ &\quad v, w := E(v), F(w) \\ &\text{ end} . \end{aligned}$$

The Rodin tool takes care of the replication and avoids generation of refinement proof obligations associated with extended events.

2.3.6 Modelling language support for proof

In the description of Event-B above at some points we have referred to the simplicity of Event-B, in particular, with respect to its support for proof as its main technique for reasoning. As a matter of fact, during the design of Event-B much attention has been paid to this. The entire notation has been designed to facilitate simple proof obligation generation and efficient retrieval of old proofs associated with proof obligations.

Simple proof obligation generation is achieved by the reduced structure of the notation. Contexts, machines, and events provide only the structure

necessary to allow the reasoning outlined above. All proof obligations are generated from the model with as little rewriting as possible. This is done in order to permit the user of Rodin easy switching between modelling and proving. It is essential that the user recognises immediately the elements of a model that make up a proof obligation, and conversely that the user can easily imagine the proof obligations associated with elements being modified.

Efficient retrieval of proof obligations is achieved by naming proof obligations systematically using labels associated with each element of a model. As an illustrative example we consider a model with two invariants

$$\begin{array}{ll} \text{inv1} & I_1(v) \\ \text{inv2} & I_2(v) \end{array}$$

and one event

$$\begin{array}{l} \text{Evt} \hat{=} \text{any } t \text{ where} \\ \quad \text{grd1 } P_1(t, v) \\ \text{then} \\ \quad \text{act1 } v := E_1(t, v) \\ \text{end} \end{array} .$$

From this model two proof obligations

$\begin{array}{l} \text{Evt/inv1 :} \\ I_1(v) \\ I_2(v) \\ P_1(t, v) \\ \vdash \\ I_1(E_1(t, v)) \end{array}$	$\begin{array}{l} \text{Evt/inv2 :} \\ I_1(v) \\ I_2(v) \\ P_1(t, v) \\ \vdash \\ I_2(E_1(t, v)) \end{array}$

are generated. Note, how easy it is to match model elements and proof obligations and how this is reflected in the naming. The naming remains when the elements I_1 , I_2 , P_1 , or E_1 are changed. Thus, it is trivial for the Rodin tool to locate proofs associated with proof obligations before the change. The speed of this is crucial for the incremental modelling approach to work. In a complex model there are usually many proof obligations but the feedback provided by the tool should depend as little as possible on the number of proof obligations. The development of the concept of witnesses also started with efficiency considerations. Without witnesses the goal of refinement proof obligations would contain a conjunction enclosed by an existential quantifier rendering decomposition of the goal according to the labelled conjuncts

impossible. By the use of witnesses the existential quantifier in the goal disappears [Hal07].

The systematic naming scheme also contributes to simplicity in the sense that the user can easily locate proof obligations when analysing specific elements of a model.

Which proof obligations are to be generated for a model is controlled by attributes associated with events. Proof obligations for convergence are associated with events by providing them with a corresponding “convergence attribute”. Similarly, event extension is available by attributing events correspondingly.

2.3.7 Differential Proving

In Event-B changes to a model are expected to occur frequently. The user is expected to improve a model in small increments. Changes happen for various reasons. Most often a model is changed because of increased understanding that has been gained through the modelling and reasoning. Sometimes a model is changed simply because of small mistakes that occur when typing formal text. And sometimes changing a model facilitates proof. When a model changes, the impact on proofs already carried out should be as small as possible. Obviously, the user should not be asked to redo a valid proof. But the same is expected concerning the tool. Proving is very time consuming and should be avoided in order to achieve better reactivity of Rodin and in order to encourage incremental modelling. We want the user to make frequent changes. Hence, the tool should manage proofs differentially only redoing a proof when its necessary. We use a number of techniques to achieve this.

Proof obligations are filtered along three stages. The first stage is purely syntactic. If a newly generated proof obligation is syntactically identical to the old proof obligation, the prover assumes validity of the old proof for the new proof obligation. This process is speeded-up by the naming scheme that permits fast retrieval of proof obligations and proofs. In the second stage, proofs of proof obligations that have changed syntactically are analysed. For each proof the tool records the hypotheses used for the proof to succeed. If none of the used hypotheses has changed and the goal has not changed, the proof is assumed valid for the new proof obligation. Finally, in the third stage, the proof is replayed attempting to rename identifiers that have been freed in quantified expressions during the proof. If this fails, the old proof has to be carried out again in full.

Chapter 3

From Requirements to Models

*M. Butler, S. Hallerstede, M. Jackson, M. Jastram, L. Ladenberger M. Leuschel,
S. Yeganehfar*

3.1 The Problem Frames Approach

3.1.1 Introduction

The focus of the Deploy project is on the development of software for computer-based systems. In developing such a system, the central—though not the only—goal is to ensure satisfaction of the functional requirements in the physical world in which the computing equipment is embedded. The Event-B approach is central to the Deploy project. It is based on building a formal model of the system behaviour, formally specifying some invariant properties that should hold over that behaviour, and proving that the model behaviour has those properties. Such an approach has obvious large advantages. Above all, it can give a high degree of confidence, amounting almost to certainty, that if the physical system conforms to the model it will exhibit the specified properties.

Computer-based systems are complex, and present many challenges. The system behaviour, at every level, must meet the needs of its ‘stakeholders’ - the system’s sponsors, relevant regulatory bodies, certification and inspection authorities, the users, the operators, and every part of the world that interacts with it or is affected by it. The developers must investigate and analyse those needs; they must design the system behaviour to satisfy them; and they must create the software that will reliably evoke that behaviour.

In the established branches of engineering, and in some areas of software engineering too, these challenges are addressed in the context of a *normal design* discipline - that is, a body of standard designs and well-established development procedures that experience has shown to be effective. Today the designer of a car or aeroplane leans heavily on precedent, adhering closely to the canons of the relevant normal design discipline.

Because normal design is specialised and comparatively well-understood by its practitioners, researchers who seek to discover or elaborate general principles and widely applicable techniques are naturally drawn to considering tasks of *radical design*, in which nothing is known a priori. In reality, every development project is partly radical and partly normal. Yet most software engineering research, unless its specific purpose is to study and analyse a particular sphere of normal design, emphasises the radical aspects of any case study or example problem. This common practice will be followed in this present section, where a lift control system will be used as an illustration of radical design. The fact that many visibly normal design aspects of commonly encountered lift systems are familiar to all readers will be simultaneously ignored and exploited.

Drawing on the lift control example, this section briefly illustrates the Problem Frames approach, and explains its rationale. The approach is not presented in detail here, but the discussion should be understandable even if the approach is unfamiliar. A basic account is available in a book [Jac01], and some further developments of the approach and its underlying principles are discussed in more recent papers [Hal07, BGJ06, Jac10, JHJ07].

The rationale of the approach is motivated by the challenges of the phase of development characterised by the title of the present chapter - the activities concerned to capture the system requirements, analyse them in their proper context, and understand how they may be satisfied. The emphasis is on the system functionality: that is, on the system's observable behaviour in operation. The problem frames approach is not a formalism or calculus, but a conceptual structuring within which appropriate formalisms and reasoning can be deployed to describe and analyse the system requirements and show how they can be satisfied. It emphasises the careful identification of distinct concerns, and their separation in the development process, always recognising their recombination as an explicit development task.

Following the account of the Problem Frames Approach, the relationship between formal and non-formal aspects of development, including their effective combination, is discussed explicitly. The discussion draws both on general principles and on the specific experience and contribution of Robert Bosch AG, an industrial partner in the Deploy project, in bringing together the use of Event-B and Problem Frames.

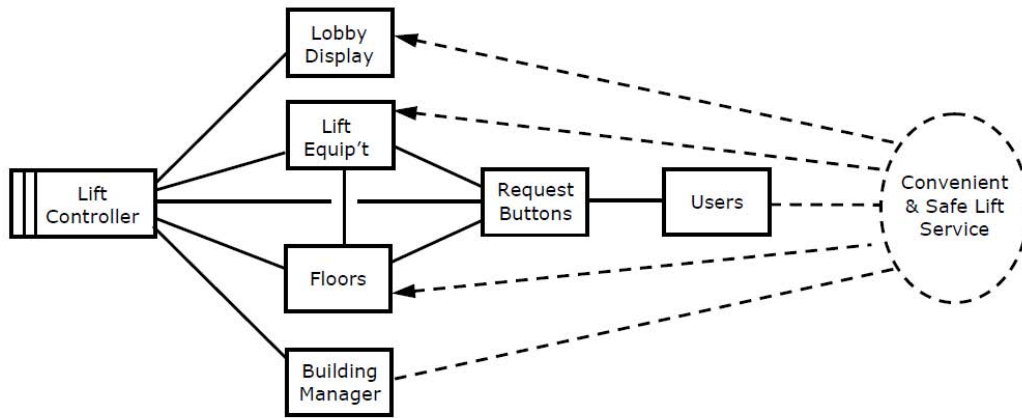


Figure 3.1: Lift Control Problem

3.1.2 What's the Problem?

As its name suggests, the approach is grounded in the notion of a development problem. A problem can be represented in a problem diagram; Figure 3.1 is an initial sketch of the lift control problem.

The diagram shows: the *machine* (the Lift Controller); the *problem world*, structured as an assemblage of *problem domains*; and the *requirement* (“Convenient & Safe Lift Service”). The machine and problem domains interact at interfaces of shared events and states, represented by the solid lines. The requirement is a condition on the problem world. For example: the lifts should respond to the users’ requests; the Lobby Display should show the current position and direction of lift travel; the lift service should be prioritised according to the specifications of the Building Manager; the system should be safe even in the presence of malfunctions of the Lift Equipment. The arrowheads show that the requirement condition constrains the behaviour of the Lobby Display, Lift Equipment and Floors; it refers to the Users and Building Manager, but does not constrain their behaviour; it does not even mention the Request Buttons.

All the problem world domains have certain given properties - for example, the way in which the hoist motor causes the lift car to travel between floors, and the construction and operation of the Lobby display. The machine must respect and exploit these properties so that the combined behaviour of the machine and the problem domains ensures satisfaction of the requirement. There are, therefore, three large subjects to be understood and described: the requirement, R ; the given properties of the world, W ; and the behaviour of the machine at its interface with the world, M . For an acceptable solution of

the problem, the entailment

$$M, W \models R$$

holds. The problem, then, is to devise a behaviour of the machine at its interface with the neighbouring problem domains: in other words, knowing W and R , to find M to satisfy the entailment.

However, this view of the problem is grossly inadequate. In reality neither W nor R is known in detail. It is therefore a central objective of the development to achieve and capture a sufficient understanding of the requirements R , and of the properties and behaviours W of the problem domains. For neither R nor W is this simply a matter of obtaining existing information from those who know it: it can be carried out only in the course of a demanding process. A phrase sometimes used in discussing the early phases of developing a computer-based system is “eliciting stakeholder requirements”. The phrase implies that the stakeholders already know what they require, and that the developers’ task is merely to elicit this information from them. This is far from the truth. First, many requirements are known only in a vague form: the system must be safe, or it must be convenient to use, or it must be secure against malicious attack. To clarify such a vague requirement is a demanding task in itself. A remarkable example [Vin93] was the desire of pilots in the early decades of aeronautical engineering for aircraft that were “stable, responsive, unsurprising and satisfactory to fly”. The desire was articulated around 1918. It took twenty five years to discover what these desirable properties really were, what aircraft behaviours would exhibit them, and how the designers could achieve that behaviour. A less dramatic example can be drawn from lift systems. Some provide request buttons only at the floors: there are no request buttons inside the lift car. Is this convenient for users? What needs does it serve well? What range of user experiences - good and bad - does it afford?

Second, the requirements of different stakeholders may conflict. The conflict may be immediately obvious, where in some operational condition two requirements are directly contradictory. Or it may be less obvious, where the requirements place conflicting demands on a common problem domain lying on the causal paths between the machine and the phenomena constrained by the requirements. Then the developers must design some compromise, or give one priority over the other.

Third, a computer-based system is necessarily intended for operation under certain assumptions in a certain context. But the operational context is not simple: it has a complex structure of subcontexts. For example, the lift system must provide reliable service while the lift equipment is functioning

correctly; but in the presence of equipment faults of various levels of severity it must maintain safety, sometimes at the expense of abandoning the service of transporting users between floors. In the event of a fire it must be operable by firefighters, who have different requirements. Licensing inspectors may have particular requirements during their inspections.

To this variety in the context and requirements corresponds a variety in the properties and behaviours of the problem world. For example, the normal operation of the lift, transporting people from floor to floor, depends on the proper behaviour of the Lift Equipment. When the machine turns the hoist motor on, the lift car rises in the shaft; and when the lift car reaches the target floor the sensor at that floor goes on, allowing the machine to detect that the motor should now be turned off. By contrast, the system function of detecting and diagnosing faults in the Lift Equipment will be concerned with its potentially faulty behaviours. Are two floor sensors ever on simultaneously? When the hoist motor is turned on, how long should it be before the sensor at the floor it is leaving goes off? When the lift car travels from floor n to floor $n + 2$, does the sensor at floor $n + 1$ go on and off as expected?

The task of formulating the requirements and problem world properties, then, is not mere elicitation. It is a task of development, even of design. The task demands the detailing, structuring and design of desired behaviours in the problem world, and of machine behaviours that will evoke them.

The problem diagram, even in this bare form, can serve as a focus for discussion and clarification. Why are the Users in the diagram? Because without them we could not take explicit account, for example, of the propensity of an impatient user to press a floor request button more than once; nor could we discuss how much time users take to enter or leave the lift car in various circumstances. The electrical power supply does not appear as a problem domain. Can we still consider faults such as power failure of the hoist motor or lobby display lights? Yes, if we treat them as a spontaneous failures of the Lift Equipment or Lobby Display. But without a Power Supply domain our analysis cannot recognise the common cause of those two faults, because the Lobby Display and Lift Equipment domains are connected only by the machine.

The interfaces of shared phenomena can be labelled, and textual annotations written to specify the shared phenomena and identify the sharing domain that controls them. For example, the machine's interface with each floor includes: the state of the sensor that detects the presence of the lift car; the events of pressing the upwards or downwards RequestButton; and the states of the motor that opens and closes the doors. An appropriate

annotation might then be:

$$\{F!sensor[f], F!upRequest[f], F!downRequest[f], M!doorOpening[f], \\ M!doorClosing[f]\}$$

where F! and M! mean respectively that the event or state change is initiated respectively by the Floor domain or by the Machine. Each Request Button is located either in a lift car or on a Floor:

$$\{= locatedAt[f, b]\},$$

and the Users press and release the Request Buttons:

$$\{U!press[b], U!release[b]\}$$

Where appropriate, a careful description is given in natural language of the problem world phenomena denoted by each formal term. This emphasis on the physical phenomena is necessary to address the risk of confusing two phenomena that are distinct but causally related. For example, in a test implementation of an automotive system the phenomenon “the throttle is open” was confused with “the accelerator pedal is depressed.” In the accident at Three Mile Island a contributory factor was a confusion between “the valve is open” and “the valve solenoid is off.”

The analysis and detailing of the problem as diagrammed can be taken as far as seems useful at this preliminary stage. However, as in any complex design task, developing the requirements and understanding how they can be satisfied will demand a decomposition of the problem into simpler constituents, and a subsequent recombination into the originally desired whole.

3.1.3 Loose Problem Decomposition

In the matter of decomposition, the Problem Frames Approach rests on two principles. First, decomposition and analysis of the requirement is accompanied by decomposing the problem into *subproblems*. Each subproblem is considered to have its own requirement, with a machine and a problem world. Second, the decomposition is *loose*: that is, constituent subproblems are identified and analysed, but they are initially analysed in isolation, their eventual recombination being temporarily ignored. The recombination then becomes an explicit development task. Loose decomposition can be likened to a preparatory phase of bottom-up design: it identifies the parts that will later be fashioned into the desired whole.

Loose decomposition is unconventional. Most current software development approaches implicitly mandate tight decomposition, in which each identified part is considered from the outset in relation to the whole and to its siblings and descendants in the decomposition. Loose decomposition is justified by several considerations. The recombination task is deferred until it can be tackled with more knowledge and confidence once the subproblems to be combined are better understood. The complexity of any subproblem or system function or feature has two sources: the intrinsic complexity of the function itself, and the additional complexity of its interaction with other functions. Loose decomposition allows these to be considered separately, and the subproblems are thereby simplified. So long as interactions are ignored, each subproblem diagram can be regarded as defining a closed system: everything relevant is included in the subproblem diagram.

Loose decomposition also improves the shape and texture of the development project. Each subproblem's investigation is a smaller commitment of resources at a time at which the development process is still largely exploratory. A subproblem investigation can be the subject of an independent work assignment in a development team. Requirements in distinct subproblems may be mutually contradictory because the subproblems concern distinct contexts or mutually exclusive problem world conditions. For example, in a train control system, one subproblem's requirement is to avoid collisions between trains by ensuring that no track segment is ever occupied by two trains simultaneously; but another subproblem's requirement is to manage the process of making up a train in preparation for its journey, joining together carriages from different trains. Loose decomposition readily accommodates this kind of contradiction: false steps in a subproblem analysis will not, in general, require rework elsewhere.

With loose decomposition in mind a tentative initial list of subproblems can be proposed. For the lift control problem, it may include the following subproblems among others:

- LiftService, providing lift service to move users safely from floor to floor;
- RunDisplay, controlling the Lobby Display to show the position and travel direction of the lift car;
- FreeFall, safeguarding the lift car from falling freely in the shaft;
- LesserFaults, handling equipment faults that are less serious than free fall; and
- DefinePriority: defining a priority scheme for LiftService.

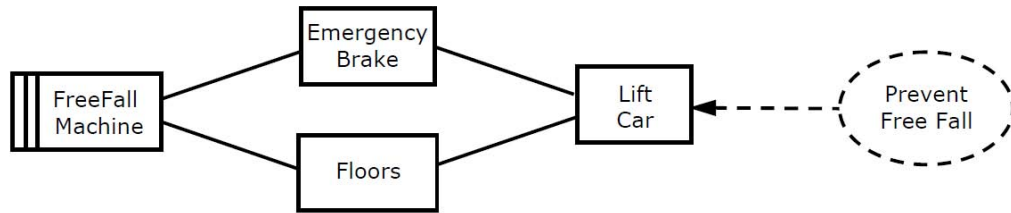


Figure 3.2: FreeFall Problem Diagram

The decomposition has two closely related goals. The whole complex problem must be reduced to an assemblage of simple subproblems; and the simple subproblems must be comprehensible in terms of the system requirements. Candidate subproblems therefore correspond to recognisable system functions and to modes and conditions of operation. The notion of a subproblem, as will be seen, supports criteria of simplicity that can help to guide the decomposition.

Figure 3.2 shows a preliminary sketch of the subproblem diagram for the FreeFall subproblem. The Lift Equipment problem domain of Figure 3.1 has been further structured. Of the resulting lift equipment parts only the Lift Car and the Emergency Brake are shown because only they are relevant here. Other Lift Equipment parts, the Building Manager, Users, Request Buttons and Lobby Display domains are all irrelevant. The required behaviour in this subproblem is that when the Lift Car is falling freely in the shaft the fall must be halted. This can be done by applying the Emergency Brake, thus stopping the Lift Car and locking its position on the vertical guides.

The diagram of a simple problem like this can be sketched quickly and easily. Like the problem diagram of the whole system, it invites questions that can focus the developers' attention on important points that merit immediate discussion, clarification, and documentation. Is it right to omit the Users? Should the Emergency Brake domain be mentioned, or even constrained, by the requirement? Because, by intent, the problem is simple, we should be able to describe the relevant given properties W of the problem world and the requirement R , and derive a machine behaviour that will ensure satisfaction of R .

Regarding the diagram as depicting an isolated system, we will be able to explain its operational principle. How will the machine and domains interact? What role must each problem domain play in contributing to satisfying the requirement? In short: How will it work? For the problem diagram of Figure 3.1, the question "How will it work?" admits no simple answer. The requirement is too complex to be well enough understood as a whole. Any attempt to explain one part of the requirement will probably be continually

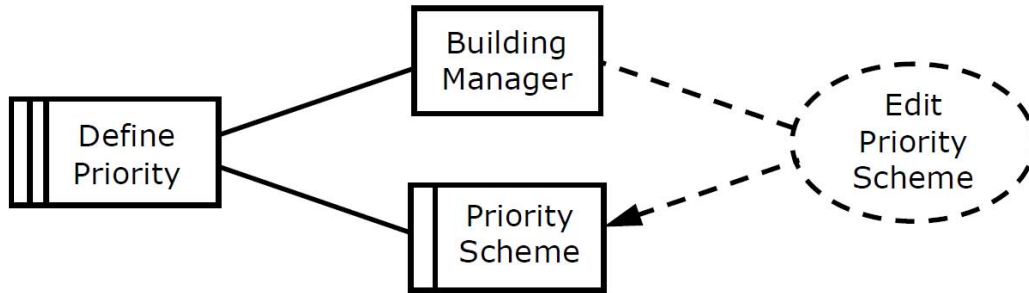


Figure 3.3: DefinePriority Problem Diagram

obstructed at every reference to a problem domain that participates also in other requirement parts.

Another subproblem is shown in the next figure. The BuildingManager is responsible for specifying priority schemes for LiftService, and was included in the problem world of the whole problem for that reason. The creation and editing of priority schemes should evidently be decoupled from their use in LiftService. To do so requires introducing Priority Scheme as a distinct instrument to mediate between the decoupled subproblems, and we call this an *instrumental decomposition*. The subproblem in which the Building Manager specifies a priority scheme is shown in Figure 3.3.

PriorityScheme is a *designed domain* - that is, a structured local variable of an undecomposed machine. It is exposed as an explicit problem domain in the subproblems of the decomposition because without it neither subproblem would make sense. We cannot understand the DefinePriority subproblem without some domain to represent the object of the manager's editing activity. On the other side of the decoupling, since the applicable scheme can vary over time, we cannot write the stipulations of the scheme into the requirement definition of the LiftService subproblem: the PriorityScheme therefore appears there too as a designed domain. The design of the PriorityScheme domain must reflect the needs of the Building Manager, the required semantics of permissible priority schemes, and the convenience of their interpretation and use in the LiftService subproblem.

Instrumental decomposition is required whenever complex information must be collected about the behaviour and state of a problem domain, transported over time or place, and analysed or summarised before use. In such cases, the designed domain provides the subproblem that uses it with a *surrogate* or *analogic model* of the problem domain. For example, the subproblem LesserFaults may be decomposed into ModelFaults and SoftLanding. ModelFaults monitors the LiftEquipment domain and detects and diagnoses

equipment faults; SoftLanding brings the car safely to the ground floor to disembark any passengers. The justification for introducing the designed domain and its accompanying decomposition is two fold. First, the resulting subproblems are greatly simplified. The designed domain is in effect a shared variable, and the separation of readers and writers has well- understood advantages. Second, the designed domain - whether in the form of a database or of an assemblage of programming objects - will certainly appear in the implemented software. A surrogate or analogic model is always an imperfect approximation of the reality, and this approximation is often a significant concern in developing the system requirements. Third, confusing the phenomena of the surrogate from the phenomena of the domain it models is a common source of difficulty.

3.1.4 Subproblem Analysis

The purpose of decomposition is to analyse a complex whole into simple parts. Loose decomposition contributes greatly to simplicity by setting aside the complexities of interaction. It is still necessary to judge whether a candidate subproblem is sufficiently simple, or demands further decomposition, or should be discarded in favour of an alternative candidate. One criterion of sufficient simplicity is the subproblem's conformance to the pattern of a problem class whose further analysis and solution is well known. The DefinePriority subproblem may be of such a kind: enough is known about the design and implementation of editors to suppose that this subproblem will offer no intrinsic difficulty that must be investigated as a part of the requirement development.

Other criteria can be more general. For example:

- Unity of Context: Different contexts of use demand different modes of operation. The system may be engaged in providing normal service, or it may be operating under fire brigade control . The context of a simple subproblem is constant over its operational life.
- Unity of Requirement: A subproblem is not simple if its requirement has the form: "Ensure P1, but if that is not possible ensure P2." This kind of cascading structure may arise in a highly fault tolerant system. The distinct levels of functional degradation can motivate subproblems.
- Unity of Problem Domain Properties: In a simple behavioural constituent each part's relevant properties are coherent and consistent, allowing a clear understanding of how the behaviour is achieved. In

a LiftService subproblem, the properties of the LiftEquipment domain are those on which the lift service function relies.

- Problem Completeness: A simple problem is closed. That is: all initiative for event occurrences and state changes is attributable to problem domains or to the machine. The system represented by the subproblem is not subject to any external influence.
- Simplicity of *Operational Principle*: In explaining how a subproblem works, it is natural to trace the causal chains in the problem diagram. An explanation of the FreeFall subproblem would trace a path over Figure 3.2 .
- From the LiftCar domain to the Floors domain: “the lift car moves between floors;”
- At the Floors domain: “lift car arrival and departure at a floor changes the floor sensor state;”
- From the Floors domain to the FreeFall machine: “the lift car movement is detected by the machine’s monitoring the floor sensors;”
- At the FreeFall machine: “the machine evaluates the speed of downward movement; excessive speed is considered to indicate free fall;”
- From the FreeFall machine to the EmergencyBrake: “if the downward movement indicates free fall the machine applies the brake”.

Satisfaction of the requirement is explained in a single pass over the causal links, with no backtracking and no fork or join. The complexity of an operational principle is reflected in the number and complexity of the causal paths in the problem diagram that trace out its explanation.

These and other similar criteria characterise extreme simplicity, and a developer’s reaction to the evaluation of simplicity must depend on many factors. It remains true in general that major deviations from extreme simplicity warn of difficulties to come. Unrecognised or neglected difficulties lead to system failures.

Even when an identified subproblem is sufficiently simple to demand no further decomposition, it may still present potential difficulties. These potential difficulties can be seen as characteristic concerns to be addressed in the subproblem analysis - often characteristic of the class of problem or the types of its problem domains. The *breakage* concern arises if the machine in a problem exerts control over a physical non-human domain such as the Lift

Equipment. It is then necessary to ensure that the behaviour of the machine will not break the domain by attempting to operate it incorrectly. In the case of the Lift Equipment, this may mean, for example, that the direction of lift car travel must not be reversed without allowing a period in which the moving parts can come to rest.

Another concern is *initialisation*. In traditional imperative program design it is recognised that accessing the undefined value of an uninitialised program variable is likely to lead to program failure. In exactly the same way, the behaviour of a subproblem machine may fail if the problem world state does not satisfy certain assumptions when the machine begins execution. For example, the LiftService subproblem may assume that when its machine starts execution the lift car is at the ground floor with the doors open and the hoist motor switched off; similarly, in a system to control access to a secure building the subproblem that monitors access may assume that initially the building is entirely unoccupied. Such assumptions belong among the assumptions of the subproblem's operational context. It is often desirable to weaken them as far as possible; but in general they cannot be made equal to *true*.

The analysis of each subproblem can be carried as far as the developers judge appropriate or necessary. For the DefinePriority subproblem it may be appropriate to design the Priority Scheme domain an abstract data type while leaving the behaviour of the editing process to be defined later. For the FreeFall subproblem it may be appropriate to specify the behaviour of the machine and the problem domains' properties very carefully, spelling out the R, W and M models in full in state machines or other formalisms.

3.1.5 Recombining Subproblems

The result of loose decomposition is an assemblage of subproblems, each analysed to a degree considered a sufficient preparation for the necessary task of recombination. Each subproblem defines a particular idealised projection of the behaviour of the whole system - that is the behaviour evoked by one execution of the system software from switching on to a quiescent halting state in which nothing further can happen until the software is switched on again. Each subproblem behaviour projection is idealised to the extent that it ignores the interactions with other subproblems; it captures, in an idealised form, not only the directly required behaviour but also the contributing behaviours of the subproblem machine and of those problem domains that the subproblem requirement does not directly mention.

Each subproblem has its own operational lifetime, from the initial activation to the termination of its machine execution. The subproblem life is, of

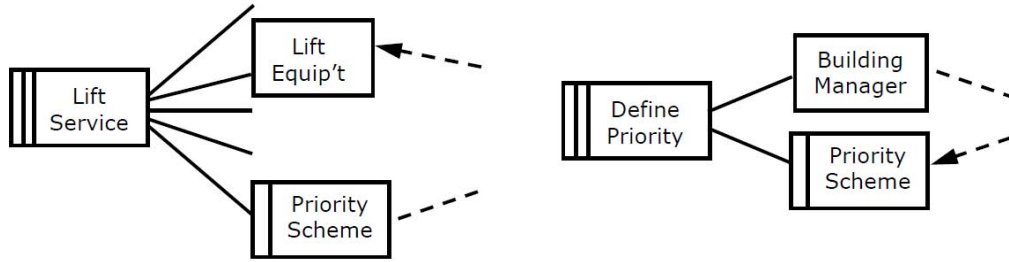


Figure 3.4: Subproblem Recombination

course, bounded by the lifetime of one execution of the whole system's software. In general, any pair of subproblems may be executed in parallel. One subproblem machine may be executed more than once, and distinct instantiations of the same machine's execution, instantiated for distinct problem world entities, may execute in parallel. This subproblem parallelism is not equivalent to any interleaving, because more than one subproblem machine may react at the same time to a single event in a common problem domain. Any resulting conflicts between subproblems - for example, initiating incompatible state changes in common problem domains - must be addressed in the recombination task.

Some particular subproblem pairs are mutually exclusive. For example, when a serious but non-catastrophic equipment fault is detected, the Soft-Landing subproblem brings the lift car safely to the ground floor lobby to disembark the current users and then terminate lift service: this behaviour is incompatible with the behaviour of the LiftService subproblem, and the two subproblems cannot operate in parallel. The RunDisplay subproblem, maintaining the Lobby Display of the lift car position, must operate from beginning to end of the system's operation. The DefinePriority

subproblem instance for one Priority Scheme can be parallel with the use by the LiftService of a different Priority Scheme.

Subproblem parallelism is a central factor in the relationships - yet to be fully explored - of the subproblem contexts and of their requirements. As these relationships are examined, more detail of the possible parallelism will become apparent, along with the need to modify some of the idealised subproblems to take account of interactions with other subproblems. Sometimes, too, it will become clear that additional subproblems are needed to mediate relationships among subproblems.

These considerations may be illustrated by the treatment of the priority scheme for lift service. Ignoring the variations in lift service for control by firefighters, licensing inspectors and maintenance engineers, the starting point for recombination may be as shown in Figure 3.4.

The diagram shows, in fragmentary form, the subproblems LiftService and DefinePriorities, with the Priority Scheme domain that was introduced to allow the two subproblems' behaviours to be decoupled. This decoupling has two obvious dimensions. Each subproblem is simplified by separating its own behaviour from the other's. Further, the writing of the priority scheme can now be separated in time from its reading by introducing a library of stored schemes as a new designed domain. The management of the library - SchemeManagement - is a fresh subproblem. SchemeManagement is interposed between DefinePriority and LiftService: at the command of the Building Manager it stores newly created or edited schemes from DefinePriority, and provides the chosen scheme to be applied to LiftService.

The need for LiftService to switch from one priority scheme to another raises a concern that is very common in subproblem recombination. This is the *switching concern*. The Lift Equipment domain, along with the Users and other relevant domains, must be switched from control by one regime to control by another. The same concern, but with different detailed demands, arises when normal LiftService must be replaced by FireBrigadeService, or InspectionService. The concern arises whether or not the old and new regimes are regarded as behaviours of distinct subproblems or distinct instances of the same subproblem. The concern does not arise within one simple subproblem, because a combination of different control regimes would have indicated that the subproblem is too complex and it would have been decomposed. In the LiftService subproblem, for example, the PriorityScheme must be regarded as a constant, excluding the possibility of changing from one scheme to another. To address the switching concern, the developers must ensure that at the switchover point the initial assumptions of the new control regime are satisfied. Depending on the details of the case in hand, this may demand use of various techniques: the switchover point may be deferred; the predecessor or successor behaviour may be modified; or a fresh subproblem may be introduced to match the predecessor's terminating problem world state to the successor's initial state.

Another example of a concern arising in subproblem recombination is *interference*. The problem completeness criterion of simplicity may demand that some problem domain be regarded as constant that in reality is subject to externally controlled change by the behaviour of another subproblem (the Priority Scheme domain is such a domain for the LiftService subproblem). The interference concern is fine-grained where the switching concern is relatively coarse-grained. An example from a different application area is a subproblem recombination in a system to administer a library. Two subproblems are BookLending, which manages loans of books, and Membership, which manages members' status. In BookLending, only members

in good standing are permitted to borrow books, and a member's status is assumed to be constant over the lifetime of a loan. The Membership subproblem, however, must manage a complex trajectory of possible states of a member. When the two subproblems are composed, the changes in members' status interfere with the BookLending behaviour. It becomes necessary to consider such obvious questions as: Can a book be lent to a member whose membership is about to expire? What about loan renewal? If a junior member becomes a senior member during the lifetime of a loan, are senior rights and obligations applied to subsequent events and states in that loan?

It may seem at first sight that the difficulties arising in subproblem recombination show that the decomposition was itself flawed, but this is not so. The difficulties are present in the problem itself. A development approach in which they do not appear explicitly is not resolving them: it is merely concealing them from sight and so leading the developers to defer their consideration until a later development stage, or to neglect them altogether. A difficulty unrecognised or neglected is a failure waiting to happen.

3.1.6 Problems, Formal Models, and Implementations

The recombination task briefly illustrated in the preceding subsection focuses on the recombination of the requirements. Incompatibilities and omissions initially ignored in the loose decomposition are addressed. A compatible set of subproblems is achieved whose requirements compose to give the requirements of the whole system, their executions fitting into an appropriate structure of sequential and parallel behaviours. The central aim of the whole process of decomposition, subproblem analysis, and recombination has been a sound understanding of the requirements development problem with all its concerns and difficulties; the central theme has been structuring to support separations of concerns.

The reasoning used in the analysis of a simple isolated subproblem is well suited to formal verification. The operational principle shows clearly the non-formal claim that is to be proved; the piecemeal traversal of the machine and problem domains by which the operational principle is explained gives a structure of lemmas about the contribution of each domain and the machine to the cooperative behaviour of the whole.

When an analysed isolated subproblem is modified to take account of its interactions with other subproblems, verification becomes more difficult. Interference by externally controlled events and state changes, excluded from the isolated subproblem, imposes arbitrary disruptions on the simple working of the operational principle; the requirement becomes less comprehensible because it is now hedged about with exceptions whose source is not fully

visible in the subproblem itself.

The resulting complete requirements structure is not well suited to formal verification of subproblem analyses and their recombination. Nor is it well suited to software or system architecture - that is, to direct elaboration to an executable implementation. The mismatch is even more severe when the implementation environment imposes its own structure - for instance, that of a hierarchical cyclic executive - on the software. This is not surprising. 'Seamless development', in which the same structure persists from the earliest development stages through to the final implementation, has obvious attractions; but it is possible only for the smallest and simplest systems. In a realistic system, any single persistent structure will be very unsuitable for one purpose or another. Reasoning within the constraints of a structure inadequate to the purpose of the reasoning is a recipe for error.

The Deploy project has adopted the Event-B method for formal reasoning about computer-based systems. As its name suggests, Event-B models the system in terms of individual events, a model consisting essentially of a structured state space and a set of guarded events whose actions cause transitions from state to state. Certain predicates on the state are specified as invariants, and by successive refinement steps the model is so constructed that these invariants hold in every state. The model events reflect both events in the computer and events in the world, and the model state reflects the combined state of the computer and world. If the physical system conforms to the model, then its behaviour will be constrained to event sequences whose successive states satisfy the invariants that have been proved to hold in the model.

An absence of heterogeneous structure makes verification easier, and by design, an Event-B model has very little structure that can directly reflect the system's functional requirements, the operational principles by which the requirements are satisfied, the properties and behaviours of problem domains, or any proposed software structure. The structure of the Event-B refinement process itself provides the only salient structure of the model. Each refinement step gives rise to a new modelling level, each submodel being related to its parent by gluing invariants and by the formal refinement relation. No distinction is made between the machine and the problem world. An invariant can represent a relationship between refinement steps, a type declaration, a given property of a problem domain, a requirement, or a constraint on the machine behaviour. Events are not grouped in any way at all - so there is no direct representation of a behaviour. They are only implicitly related by the references in their guards and actions to the global state.

This lack of structure in an Event-B model not only simplifies the formal processing necessary for proofs; it also allows it to form an intermediate

representation between a problem-oriented structure of the requirements and a radically structure of the software oriented towards implementation. The product of the development project can then include these three structures and the relationships among them:

- Requirements: a structuring of the desired system behaviour, clarifying the requirements and showing how the software behaviour will ensure their satisfaction.
- Verification: a structuring of the system behaviour in a model that allows mechanical verification of some properties and informal validation - by animation and other means - of some others.
- Implementation: a structuring of the verified software behaviour to allow efficient execution on the system's computational infrastructure.

Here we focus only on the relationship between the requirements and verification structures.

3.1.7 From Requirements to Models

In general, the requirements structure must be developed ahead of the verification structure. The appropriate process must vary according to the system to be developed, its degree of criticality, and the project resources and circumstances. In one project it may be right to complete a fully detailed requirements analysis, including explicit models of machine and problem domain behaviours, for submission to intense scrutiny by the most important stakeholders before attention is turned to verification. In another the loose decomposition of requirements may be exploited to allow verification of some subproblems and clusters of subproblems to begin while other parts of the requirements have not yet been analysed or even structured. The depth of detail in problem analysis will vary between and within systems. For some subproblems the verification model will be as close as possible to a systematic derivation from a fully detailed subproblem analysis; for others it may be almost a free creation *ab initio*, loosely guided by a problem diagram with annotated interfaces and a roughly stated requirement. We do not attempt to recommend a systematic process, but rather to point out some structural concerns, to identify their origins in the requirements structure, and in some cases to mention possible ideas for addressing them.

Among these concerns are:

- Simple subproblems. A subproblem in its simple and isolated form may be independently verified. Independent verification is likely to be of most value for a critical subproblem such as the FreeFall subproblem.
- Sequential behaviour. If sequential behaviour of a subproblem machine, a problem domain, or a requirement, has been expressed in a finite automaton or in temporal logic formulae, the states - implicit or explicit - over which the behaviour is defined must all be explicitly modelled in the Event-B state. They can then be accessed and assigned in the guards and actions of model events.
- Behavioural requirements. Some requirements may be conveniently expressible in invariants. For example, in the DefinePriority subproblem the effects of editing commands must respect the defined syntactic and semantic properties of the Priority Scheme domain; invariants seem to offer an appropriate medium to model this requirement. Safety properties of behavioural requirements can be modelled, but liveness properties present particular difficulties.
- Operational principles. Because Event-B abstracts from control initiative, it is hard to formalise operational principle - the explanation of 'how it works'. The Event-B approach - realistically but perhaps surprisingly - advocates the use of animation, which can contribute usefully to recognising the operational principle and informally corroborating that it has been realised.
- Modified subproblems. When a simple subproblem has been modified to take account of its interactions with other subproblems, the verification task may be significantly harder. The requirement, and the behaviour to satisfy it, is now complicated by the intrusion of externally
- Recovering a simple subproblem. If the simple isolated version of a modified subproblem has not been preserved in the documentation of the requirements analysis, its verifiable behaviour may be recoverable from the modified version by adding invariants to exclude externally controlled state changes. The requirement of the recovered simple version may be verified, and its behaviour informally explored by animation.
- Distributed event responses. When two or more subproblems are considered together, and distinct subproblem machines respond to the same event in a common problem domain, a difficulty arises in modelling their joint response. Essentially, an Event-B model of a problem

domain event brings together all the effects caused by that event in the action part of the event specification. In the requirements analysis, by contrast, these effects are distributed into the distinct subproblem behaviours that recognise and respond to that event. It is not yet clear how this difficulty can be addressed in its general form. In a particular system where cases of distinct responses to a common event are few, the cases may be dealt with by merging the models of the responses.

- Behavioural system structure. To represent multiple subproblem behaviours in one model it is necessary to recognise and model the multiplicity of contexts and the changing population of subproblem operational lifetimes. The complete system operational lifetime may be represented in the model state by an activation set A whose elements correspond one-for-one to the system's subproblems. Activation of a subproblem s is then represented by inserting the corresponding element s into this set; termination, whether externally imposed or caused by an internally reached halt, is represented by the removal of the element s .
- Activating and terminating subproblems. The requirement of a subproblem is in force, and its behaviour is realised, only during its activation lifetime. Therefore if I is an invariant associated only with subproblem s , then when s is modelled in combination with another subproblem t I must be replaced by $s.A \wedge I$. Similarly, the condition $s.A$ must be conjoined with the guard of every event in the behaviour of s .
- It seems unlikely that a complete verifiable model of a large realistic system can be developed. Very small systems, and individual features of the kind that are common in automotive systems, are better candidates for complete verification.

3.1.8 Acknowledgements

The Problem Frames Approach is a work in progress. This material has benefited greatly from many discussions with Cliff Jones and Manuel Mazzara of Newcastle University and with Rainer Gmehlich, Felix Loesch, Katrin Grau and Christine Rossa of Robert Bosch AG. Extended independent discussions with Anthony Hall have also been a source of illumination and a stimulus of valuable ideas.

3.2 Structuring Functional Requirements of Control Systems to Facilitate Refinement-based Formalisation

Good requirements structure can greatly facilitate the construction of formal models of systems. This section describes an approach to requirements structuring for control systems that aims to facilitate refinement-based formalisation. In addition to the well-known monitored and controlled phenomena used to analyse control systems, we also identify commanded phenomenon reflecting the special role that an operator plays in system control. These system phenomena guide the structure of the requirements analysis and documentation as well as the structure of the formal models.

We model systems using the Event-B formalism, making use of refinement to support layering of requirements. The structuring provided by the system phenomena and by the refinement layers supports clear traceability and validation between requirements and formal models. As a worked example, we structured the requirements of an automotive lane departure warning system using this approach. We found missing requirements through this process and we evolved the requirement document through domain experts' feedback and formal modelling.

3.2.1 Introduction

Control systems are usually complex as they continually interact with and react to the evolving environment. Because of the complexity of these systems, constructing and structuring their functional requirement documents (RD) can be a time consuming process. In addition, their RD may not be clear and complete for developers of the system. However, since these systems are usually used in life critical situations it is essential to have a comprehensive RD to help with the improvement of safety and reliability of the system.

Formal methods are mathematical based techniques used for specification and development of systems as well as verifying their properties [Win90]. Modelling using formal methods is known to improve system understanding and thus help to find missing and ambiguous requirements. However, one difficulty of using formal modelling is formalising an informal RD.

We propose an approach to construct the RD of control systems incrementally to help with understanding the system requirements and to facilitate the process of formal modelling. This approach consists of three stages and is based on monitored, commanded and controlled (MCC) phenomena introduced in [But09c] as an extension of Parnas' 4-variable model [PM95b].

In the first stage an RD is constructed and structured incrementally through iterations, as our understanding of the system improves (i.e. by considering the requirements in more depth). The second stage involves modelling the RD in a step-wise manner by using refinement. Here, requirements are layered and each layer is modelled in one refinement level. The third stage of this approach deals with any identified missing and ambiguous requirements by revising the RD and the model.

This approach also provides the means for validating a model against its RD in order to ensure that the model is an accurate representation of the system's requirements. This validation also facilitates the traceability between a model and its RD.

As a worked example, we structured the requirements of an automotive lane departure warning system (LDWS) using the proposed approach. Requirements of this system are evolved in three phases. In the first phase we produce and structure the RD of the LDWS based on information in the *public domain*. In the second phase, the generated RD is discussed with *domain experts*. In the third phase, the RD of LDWS is *formally modelled* using Event-B formal language. Also, as will be discussed any changes in requirements, i.e. identified missing and ambiguous requirements, are applied to both the RD and the Event-B model.

3.2.2 Guidelines for Modelling Control Systems

The guidelines outlined in [But09c] can be used for formal modelling of control systems. The formal models consist of variables and guarded actions (events) and control systems consist of plants, controllers and in some cases operators who can send commands to the controller, shown in Figure 3.5¹.

The modelling steps suggested in this guidelines are based on the four-variable model of Parnas [PM95b]. Variables shared between a plant and a controller, labelled as 'A' in Figure 3.5, are known as environment variables and are categorised into *monitored variables* whose values are determined by the plant and *controlled variables* whose values are set by the controller. There are also *environment events* and *control events* which update/modify monitored and controlled variables respectively. The other two variable categories of the four-variable model are *input* and *output*. In [But09c] it is suggested that these are not used in abstract formal model; instead [But09c] provides patterns for introducing them as refinements.

If a system involves operators, according to [But09c] in addition to the phenomena introduced in the four-variable model, phenomena shared be-

¹The diagram uses Jackson's Problem Frame notation [Jac01].

tween controller and the operator can be identified. These phenomena, labelled as ‘B’ in Figure 3.5, are represented by *command events* which are the commands given by an operator and *commanded variables* whose values are determined by command events and can affect the way other events behave.

In [YBR10] a cruise control system is modelled following MCC guidelines. In addition [YBR10] shows that modelling based on the MCC guideline helps to have a more structured process of modelling and refinement for a control system.

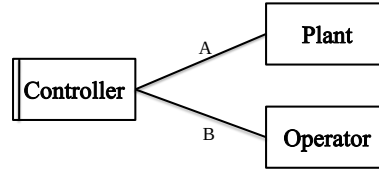


Figure 3.5: A control system.

3.2.3 Overview of the Proposed Approach

Modelling guidelines represented in [But09c] requires the modeller to identify the MCC phenomena of a system before the commencement of the modelling process. This inspired us to propose an approach for structuring RD and modelling using MCC phenomena. This approach comprises of three stages, which are shown in Figure 3.6.

Notice that we use the term *phenomena* when we deal with (informal) requirements of a system and the term *variable* when we model the system formally. Also as our focus is not on requirement elicitation methods, it is assumed that the textual RD of the control system exists.

Stage 1: Structuring Requirement Document

The following are the steps suggested for structuring the existing textual RD into monitored, commanded and controlled sections:

1. *Identify the system’s MCC phenomena* based on its textual RD. An example is the monitored phenomenon *speed* in LDWS.
2. *Organise RD* into three monitored (MNR), commanded (CMN) and controlled (CNT) sections, each representing requirements of the corresponding phenomenon. If the requirement refers to
 - only one phenomenon, it will be moved to the relevant MCC section.

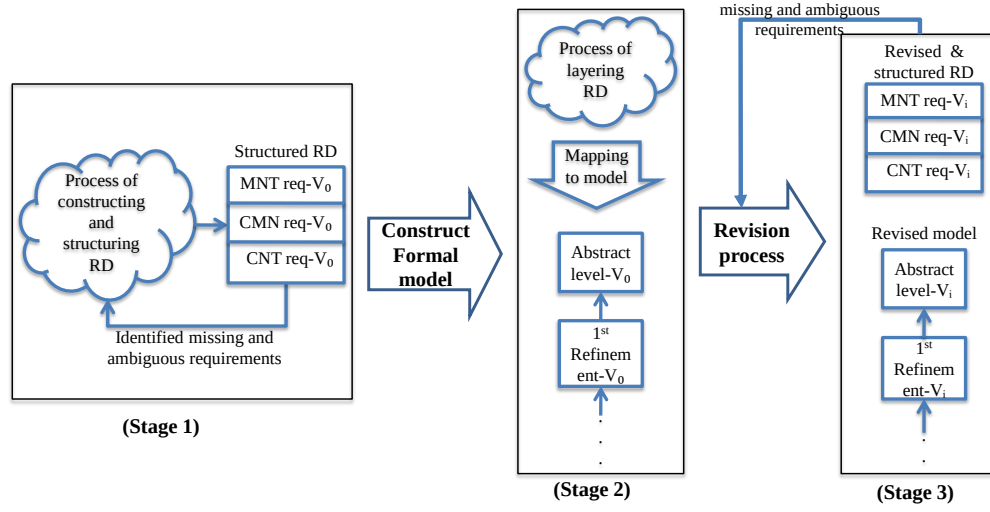


Figure 3.6: Overview of the three stages. Stage 1 - Structuring RD; Stage 2 - Layering structured RD and designing the initial formal model; Stage 3 - Revising the RD and the model.

- more than one phenomenon, but of the same type (e.g. they all are monitored phenomena), the requirement will be added to the corresponding section.
 - more than one phenomenon, but of different type, then it is designer's judgment which section is mostly appropriate.
3. *Add unique ID labels.* Every ID starts with the section that the requirement belongs to (i.e. MNR, CMN, CNT), followed by a unique number for that requirement.
 4. *Revise RD* to accommodate any identified missing or ambiguous behaviour of the system. The revision step involves going back to Step 1 to identify any phenomena that the new requirement represents and then adding the requirement to the appropriate section.

The last step helps one to seek the most obvious MCC phenomena in the initial structuring of the RD and improve it incrementally through iterations. This iteration is shown in Figure 3.6-Stage 1. Section 3.2.5 represents structuring the RD of an LDWS using these steps.

Stage 2: Layering Requirements and Designing the Initial Models

In order to deal with the complexity of a control system, our aim is to use refinement to introduce system requirements in a step-wise manner. However,

deciding on how to layer requirements and what to model in each levels is usually difficult.

We propose to overcome this problem by modelling one **feature** and the minimum number of requirements essential for this feature to be meaningful in one level of refinement. A feature is usually one of the MCC phenomena of the system. However, sometimes a phenomenon is interrelated to other phenomena and thus they should be modelled simultaneously. Examples of features for the LDWS are phenomena *warning* and *status*.

We also suggest to focus on the **main role or behaviour** of the system, which usually corresponds to a controlled phenomenon, in the most abstract level. If the system has more than one role, it is the modeller's judgment to choose the most important role to be initially modelled. This means the abstract model will focus on the role of the control system, while the rest of the requirements will be elaborated into the model through refinement levels. For instance, the main behaviour of an LDWS is to issue *warnings* and this is modelled in the abstract level. After that in the first refinement the phenomenon *status* is introduced. Section 3.2.6 describes this stage in more details through the LDWS example.

Stage 3: Revision of RD and Formal Model

Modelling a system formally can result in finding missing and ambiguous requirements of the system. We suggest to handle these requirements similarly to the revision step in Stage 1, where phenomena of new requirements are identified and based on them requirements are added to the corresponding MCC sections of the structured RD. However, in addition to revising the RD, it is necessary to update the formal model. This is because the RD and the model should be kept consistent to help with the process of validation and traceability.

If a new requirement is related to any of the previously modelled phenomena (modelled in Stage 2), this requirement can be modelled in the same refinement level as its phenomenon. For instance if the new requirement gives further information about the main behaviour of the system, which is modelled in abstract level, we update this level. However, if the newly identified requirement has no effects on any levels of the current formal model, for instance if the requirement introduces a new phenomenon, it can be introduced to the model in a new refinement level.

As shown in Figure 3.6, Stage 3 can be iterated meaning that as long as new missing or ambiguous requirements are identified, the RD and the model of the system should be revised. This stage is explained further in Section 3.2.7 using the example of LDWS.

3.2.4 An Overview of LDWS

LDWS is a driver assistance system which receives camera observations of the lane and uses this information to warn the driver of a lane departure, when the car is travelling above a certain speed. One way to detect the car departing the lane is by estimating the car's current position in the lane using lane detection algorithm on camera's observation [RME00].

In order to warn the driver before the vehicle crosses the lane, a virtual lane width which is inside the lane boundaries is assumed. This virtual lane, called the earliest warning lines (EWL), is determined by the LDWS based on the speed of the car. When the vehicle is within the earliest warning lines, the LDWS does not issue any warnings. This area is called the “no-warning zone”, and the area pass EWL is the “warning zone” [Fed05], shown in Figure 3.7.

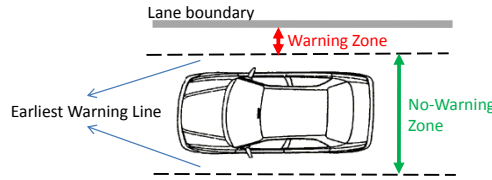


Figure 3.7: Warning and no-warning zone for an LDWS.

3.2.5 Stage 1: Constructing and Structuring RD of LDWS

In this section we discuss some of the requirements of the LDWS which are structured according to the first stage of the proposed approach, Section 3.2.3. The RD which is produced based on the information available in the public domain [Fed05, RME00, PMGB05] is outlined in Section 3.2.5. This RD then is modified in Section 3.2.5 according to the feedback received from the domain experts.

First Version of Requirement Document

To structure the RD we first examined the public sources to identify the MCC phenomena. The identified **monitored phenomena** are *position* of the car relative to the centre of the lane, *lane width* and current car *speed* which also determines the EWL. The identified **commanded** and **controlled phenomena** are respectively *status* of the LDWS which is set through a switch button and *warning*. Requirements related to these phenomena are organised

Table 3.1: First version of the structured RD of the LDWS.

Req ID	Requirement Description
MNR1	LDWS should detect the earliest warning line (EWL) and vehicle position relative to visible lane boundaries based on the lane width and car width.
MNR2	LDWS should track lane boundaries where lane markings are clearly visible in daylight (sunny/cloudy), night times and twilight (sunrise/sunset) lighting conditions.
MNR3	The width of the “warning zone” depends on the speed of the car. The higher the speed of the car the closer the earliest warning line (EWL) to the centre of the lane.
CMN1	LDWS can be switched on and off by the driver through a single button.
CNT1	LDWS should issue a warning when the vehicle has left the no warning zone (has crossed the EWL), and is entering the warning zone.
CNT2	When LDWS is on it starts its role provided that the car speed is greater or equal to a certain speed.

into MNT, CMN and CNT sections using appropriate identifiers for every requirement as shown in Table 3.1.

Second Version of Requirement Document

Discussing the first version of the LDWS’s RD with domain experts from GM India Science Lab resulted in identifying some missing requirements. One of these requirements was that the driver can change the EWL by setting the offset they wish to have from the lane boundaries.

We retook the structuring steps and as the result the commanded phenomenon *offset* was identified. Thus, the three requirements related to offset, shown in Table 3.2, are added to CMN section. Notice that we refer to requirement MNR3 in CMN4 to show that these requirements are related.

Table 3.2: Second RD - Requirements are added based on experts' feedback.

Req ID	Requirement Description
CMN2	The driver can set the offset they want to have from either side of the lane boundaries through two buttons which are responsible for increasing and decreasing the distance.
CMN3	The offset is always within a certain positive range.
CMN4	In addition to speed (MNR3) the width of the “warning zone” or earliest warning line depends on the offset from the lane boundaries. The greater the determined offset the closer the EWL to the centre of the lane.

3.2.6 Stage 2: Layering Requirements and First Model of LDWS

The initial formal model of LDWS was produced based on the second RD. We have used Event-B formal language which also provides supports for refinement. The process of modelling starts with identifying requirements necessary for constructing the abstract level. After building the abstract model, the remainder of the RD is introduced in refinement levels.

Event-B and its Tool

We only focus on two elements of an Event-B [Abr10] model. Firstly, *variables* and secondly *events*. An event consists of two elements, *guards* which are predicates defined for describing the conditions need to hold for event occurrence, and *actions* which determine the changes of state variables. An event becomes *enabled* if its guards hold. One of the advantages of Event-B is its open source tool, known as Rodin, which provides automatic proof and a wide range of plug-ins [RODa].

Requirements for Modelling the Abstract Level

As mentioned in Section 3.2.3, Stage 2 involves layering and modelling requirements based on the features of the system. In this stage, each feature is modelled in one level of refinement. Also, the main behaviour of the system (i.e. the main controlled phenomenon) is modelled in the most abstract level. Examining the structured RD of the LDWS shows that the requirement **CNT1** represents the main role of this system; “issuing warnings when

the car has left the no warning zone”. Thus, the controlled phenomenon *warning* is to be modelled at the abstract level.

After this we identify any interrelated phenomena, usually monitored and commanded requirements (MNR, CMN), which are vital for modelling *warning*. Thus, requirements related to crossing EWL should also be modelled at this level. These are firstly, the requirement **MNR1**, since the controller should receive the monitored phenomena *car position* and *lane width* in order to decide whether or not the EWL is crossed. Secondly, **MNR3** and **CMN4**, since the position of the EWL depends on the monitored phenomenon *speed* and the commanded phenomenon *offset*.

The next step is to identify requirements which describe the limitations and restrictions of the identified phenomena, or requirements which show how these phenomena affects/restricts the system. Based on this we need to model firstly **CNT2**, which describes the restriction imposed on *warning* by the system. The second is **CMN3**, as it gives details about restrictions of *offset*.

Finally we identify any requirements which represent state changes of the identified phenomena. This results in identifying **CMN2**, since it represents how *offset* can be modified. Notice that in Event-B requirements which are identified as restrictions of phenomena are usually modelled as guards for events or as types of the phenomena. Also, requirements which define state changes are usually modelled as actions of events.

Modelling Abstract Level

In the previous section, requirements and phenomena for modelling the abstract level were identified. In this section we start modelling the abstract level by representing each of these phenomena as a system variable. Also according to [But09c], the corresponding *environment*, *command* and *control* events for every variable is defined.

At this level, the monitored variables are *speed*, *laneWidth*, and *carPosition* (the difference between the car centre and the lane centre). The LDWS also needs to know the width of the car, modelled as the constant *carWidth*, in order to detect the value of *carPosition*. Environment events which are responsible for modifying monitored variables are defined to simply set the monitored variables non-deterministically through a parameter (MNR1 and MNR3).

The commanded variable modelled at this level is *offset* (CMN2). Since the value of *offset* is within a specific range (CMN3), two constants called *offset_LB* and *offset_UB* are defined to represent the lower and upper bounds of this variable. The command events which modify the value of *offset* are

IncreaseOffset and *DecreaseOffset*. Also, the controlled variable *warning* is defined as a Boolean variable. The control event *IssueWarning*, shown in Figure 3.8a, is defined to set *warning* to TRUE when the car crosses the EWL (CNT1).

To model crossing EWL, we firstly define a function named *EWL_Func* which returns the distance of the EWL from the lane boundaries based on the car *speed* and the *offset* (MNR3 and CMN4). This function is defined as $EWL\_Func \in \mathbb{N} \times offset\_LB..offset\_UB \rightarrow \mathbb{N}$, meaning that for every possible tuple of *speed* (of type $\mathbb{N}$) and *offset* (within the range $offset\_LB..offset\_UB$) there is a value for the *EWL_Func*. We assume that the return value of this function, which represents the position of EWL, are provided. Based on this function *grd3* in Figure 3.8a is defined to model that event *IssueWarning* will be enabled when the car passes the EWL. In addition, CNT2 is modelled by defining a constant *minSpeed* and adding *grd1* in Figure 3.8a.

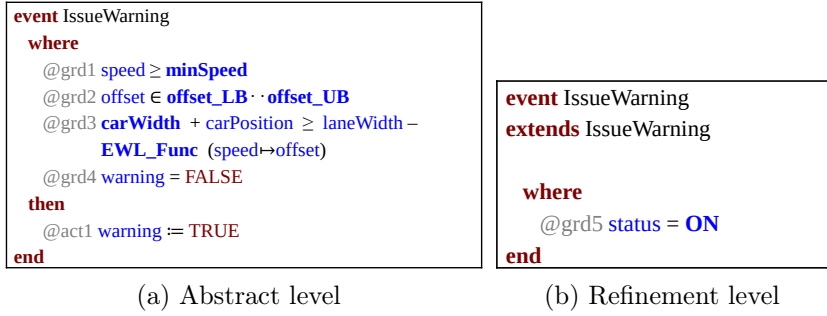


Figure 3.8: Control event *IssueWarning*.

First Refinement

In this level of refinement we focus on the feature *status* which is a commanded phenomenon (CMN1). The process of identifying requirements corresponding to this phenomenon is similar to the process mentioned in Section 3.2.6 for the controlled phenomenon *warning*.

Since no other phenomenon is interrelated to *status*, at this level of refinement only *status* is introduced. Also, requirement CNT2 is added to the model at this level, as it represents a restriction imposed by the phenomenon *status* on the controller.

This phenomenon is modelled by defining the variable *status* and command events *SwitchOn* and *SwitchOff* which set *status* variable from OFF to ON and vice versa (CMN1). Also, CNT2 is modelled by adding *grd5* to the control event *IssueWarning* as shown in Figure 3.8b.

3.2.7 Stage 3: Revision of RD and Model of LDWS

Modelling requirements formally helped us to find some of the missing and ambiguous requirements. So, based on Section 3.2.3, in Stage 3 we revise the RD and the model of the LDWS.

Missing Requirements and Revision of the RD

Two of the identified missing requirements are discussed in this section. The first is that we realised requirements related to the situations where a car *has crossed* the actual lane boundary and is travelling on the boundary should be differentiated from when the car has crossed the EWL and therefore is *about to cross* the boundary. As shown in Figure 3.9, this is mainly because of limitations of the camera's field of view which can result in detection of only one boundary.

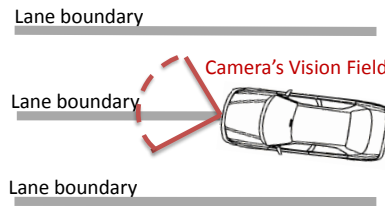


Figure 3.9: Limited field of vision for a camera when car is travelling on lane boundary.

We retake the structuring steps of Stage 1, Section 3.2.3, to add relevant requirements to the structured RD. Firstly, the LDWS should detect that the car has crossed the boundary. Thus, the requirement MNR4 is added to the RD. Secondly, the LDWS should issue warnings if crossing boundary is detected. This resulted in producing requirement CNT3, Table 3.3.

The other missing requirements are about the situations under which the LDWS should stop the warning process. Examining the LDWS showed that the process of issuing warnings should finish when the driver steers away from the lane boundaries and thus from the EWL. Here, the car is back within the lane and any issued warning should be stopped. Taking the structuring steps resulted in adding requirement CNT4 to the RD. Another situation where warning should stop is if the LDWS is switched off by the driver. This requirement is shown as CMN5 in Table 3.3.

Table 3.3: Third RD - Requirements are added based on Formal Modelling.

Req ID	Requirement Description
MNR4	LDWS should detect when the car has crossed the lane boundaries and is travelling on the boundary.
CMN5	If LDWS is issuing the warning and the driver switches the system off, the warning signal should be stopped.
CNT3	In addition to (CNT1), LDWS issues warning when it detects that the car has crossed the lane boundaries and is travelling on the boundary (MNR4).
CNT4	LDWS should stop the warning signal when the system is performing (CNT2) and the warning signal has been issued but the driver steers away from the boundaries and therefore the car remains within the no warning zone.

Adjusting Abstract Model

To identify which of the newly found requirements of Table 3.3 need to be added to the abstract model, we retake the requirement selection process of Stage 2. As the phenomenon *warning* was modelled in the abstract level, we need to consider whether any of the new requirements are related to this phenomenon. The examination of the new RD shows that requirements **CNT3** and **CNT4** need to be introduced at this level. In addition, **MNR4** should be modelled here, because the monitored phenomenon *crossing lane* is required for modelling CNT3.

Therefore, the abstract model is modified by defining *crossingLane* as a Boolean monitored variable which is TRUE if the car has crossed the lane boundary. CNT3 shows that there are two cases where the LDWS should decide on issuing warnings. Firstly when the car is *about* to cross the lane boundaries because it has crossed the EWL (CNT1). Secondly, when the car *has* crossed the lane boundary (CNT3). These are modelled in the two control events, *IssueWarning_CloseToBound* and *IssueWarning_CrossingLane* respectively, Figure 3.10. The requirement CNT4 is modelled by introducing the new control event *FinishWarning*. At this level of abstraction this event has a guard as *warning = TRUE* and the action *warning := FALSE*.

<pre> event IssueWarning_CrossingLane where @grd1 $speed \geq minSpeed$ @grd2 $warning = FALSE$ @grd3 $crossingLane = TRUE$ @grd4 $status = ON$ then @act1 $warning := TRUE$ end </pre>	<pre> event IssueWarning_CloseToBound where @grd1 $speed \geq minSpeed$ @grd2 $offset \in offset\_LB \dots offset\_UB$ @grd3 $carWidth + carPosition \geq$ $laneWidth - EWL\_Func(speed \mapsto offset)$ @grd4 $warning = FALSE$ @grd5 $crossingLane = FALSE$ @grd6 $status = ON$ then @act1 $warning := TRUE$ end </pre>

Figure 3.10: Control events *IssueWarning_CloseToBound* and *IssueWarning_CrossingLane*

Adjusting First level of refinement

As mentioned, requirements of the commanded phenomenon *status* were introduced in the first refinement level. Examining the new requirements of Table 3.3 show that CMN5 needs to be modelled at this level. This is done by defining two *SwitchOff* events. The event *SwitchOff1* sets variables *status* to OFF and *warning* to FALSE, if it was previously TRUE, while *SwitchOff2* sets *status* to OFF and has the guard *warning = FALSE*.

3.2.8 Validation of the Model

Validation of the model against its RD can be done by adding a *validation column* to the right hand side of the structured requirement tables. Every requirement is then validated by adding the elements of the model, such as an event or a variable, which represent that requirement to its validation column. Also as quick reference, we refer to the level in which the requirement was modelled just after the ID of the requirement. Table 3.4 shows validation of some of the LDWS requirements against the model. For instance, requirement CNT1 is modelled in the abstract level using the following elements:

1. the controlled variable *warning*;
2. the control event *IssueWarning_CloseToBound*;
3. and showing that the car has entered the warning zone through the guard $carWidth + carPosition \geq laneWidth - EWL\_Func(speed \mapsto offset)$ in the control event *IssueWarning_CloseToBound*.

It is important to mention that the process of validation should take place at the end of every modelling step. This means as well as modelling RD, validation should be done incrementally. Thus, if a requirement is modified,

Table 3.4: Validation of the requirements against the model.

Req ID	Requirement Description	Validation Column
MNR1 ABST	LDWS should detect EWL and vehicle position relative to visible lane boundaries based on the lane width and car width.	Monitored variable: <i>carPosition</i> & <i>laneWidth</i> ; Constant: <i>carWidth</i> ; Environment event: <i>UpdateCarPosition</i> & <i>UpdateLaneWidth</i> .
MNR2	LDWS should track lane boundaries where lane markings are clearly visible in daylight (sunny/cloudy), night times and twilight (sunrise/sunset) lighting conditions.	Not Considered, since we are not concerned with requirements related to the performance of the camera and image processing unit.
CMN1 Refine	LDWS can be switched on and off by the driver through a single button.	Set: STATUS= {ON,OFF}; Commanded variable: <i>status</i> ; Command event: <i>SwitchOn</i> & <i>SwitchOff</i> .
CMN2 ABST	The driver can set the offset they want to have from either side of the lane boundaries through two buttons which are responsible for increasing and decreasing the distance.	Commanded variable: <i>offset</i> ; Command event: <i>DecreaseOffset</i> & <i>IncreaseOffset</i> .
CNT1 ABST	LDWS should issue a warning when the vehicle has left the no warning zone (has crossed the EWL), and is entering the warning zone.	Controlled variable: <i>warning</i> ; Control event: <i>IssueWarning_CloseToBound</i> ; Guard: $carWidth + carPosition \geq laneWidth - EWL\_Func(speed \mapsto offset)$.

the model and the validation column both should be updated to keep them consistent with the RD. Also, not always the entire RD is modelled, which means the validation column may only contain the reason for not modelling the requirement rather than the elements of the model. This is the case for

the requirement MNR2, Table 3.4.

3.2.9 Related Work

In this section we look at some related works and compare them to our proposed approach.

Concretization and Formalization of Requirements

[FHP⁺05] has provided some guidelines for concretization and formalization of requirements of embedded systems. In formalization part, four steps have been suggested. *Identification* to produce an RD; *Normalization* to construct a glossary of terms that requirements use; *Structuring* to organise RD based on their “contents in a taxonomy”, such as car speed; *Formalization* to formalise the structure, behaviour, interaction and data of the system.

In [FHP⁺05] grouping requirements is based on “different aspects”, while we represent guidelines based on MCC phenomena for an engineer. In addition, our proposed approach allows one to revise the structured RD and construct a formal model incrementally. This means identified missing/ambiguous requirements can be addressed in the forthcoming iterations, while [FHP⁺05] does not tackle this issue. Also, we proposed a way for layering requirements which facilitates the use of refinement-based modelling, while this is not specified in [FHP⁺05].

HJJ

In the HJJ approach [HJJ03] the specification of control systems is initially based on the system view rather than the software view. In this respect our approach has a similarity to HJJ. In the HJJ approach the focus is to model the environment and requirements while also the properties that the control system relies on are captured as “rely condition”.

In HJJ all requirements of a system are dealt with in one step of specification, while our approach uses refinement. Also, we differentiate between monitored, commanded and controlled phenomena which assists with structuring the RD and mapping it to a formal model.

Requirement Tracing based on WRSPM

[JHLR10] introduces an approach for tracing requirements to an Event-B formal model. This approach is based on WRSPM [GGJZ00] (World, Requirement, Specification, Program and Machine) which distinguishes between phenomena, system’s state space, and artifacts which represent constraints.

The method of [JHLR10] for traceability involves taking a requirement and identifying phenomena and artifacts of the environment and system for that requirement. The identified phenomena and artifacts are then modelled and traceability information is provided.

Both our approach and [JHLR10] are concerned with formal modelling of informal requirements and providing traceability. While [JHLR10] is more focused on traceability between an RD and the formal model, the approach represented in this section is more on structuring and formal modelling. Also, we provide some guiding steps for layering requirements for a refinement-based modelling, while this is not specified in [JHLR10].

SCR

Our approach shares features with the SCR (Software Cost Reduction) method [HJL96]. SCR is a formal method for specification of control systems with a tabular notation.

Like our approach, SCR is based on the four-variable model. In addition to these four variables, the SCR uses mode classes (the system states), conditions (predicates of system states), and events (represent changes in system variables and mode). The SCR method does not have commanded phenomena though these can be represented as monitored variables.

Our experience is that distinguishing monitored and commanded phenomenon facilitates requirements elicitation as they serve distinct roles. SCR is more a specification method, while in this section we focus on structuring the RD as well as specification and traceability.

In SCR an engineer is required to identify the monitored, controlled, input and output variables of the system. However, we have a system-level view on the behaviour of the controller. Therefore, we focus on monitored, controlled and commanded variables in the more abstract models and introduce input and output variables in refinement levels.

3.2.10 Future Work and Limitations

This approach can be improved and developed further by experimenting its application in other case studies. Also, the current approach focuses on the functional RD and we are yet to deal with some of the challenges presented by non-functional requirements. In addition to improving the approach, part of our future work involves developing a more complete formal model of the LDWS. Examples of requirements which can be considered in the future work are timing and fault tolerance requirements. Our other future work

involves evolution of the passive LDWS to an active lane centring system and examining the evolution of the requirement document in this case.

3.2.11 Conclusion

We discussed the evolution of the requirement document of the LDWS through domain expert's feedback and modelling using Event-B. The MCC modelling guidelines [But09c] inspired us to structure the requirement document based on monitored, controlled and commanded phenomena. Also some criteria for layering the requirements and mapping informal requirements to a formal model were provided. We followed the proposed approach to structure and model the RD of an LDWS. Some of the advantages provided by following the proposed approach are:

- Improving the requirement document by gathering and structuring the requirements incrementally in iterations.
- Facilitating the process of validation of the model against the requirement document and therefore helping with traceability between the model and requirements.
- Traceability enables us to maintain the requirement document and the model consistent in an easier and more manageable style.

The process of formal modelling of the LDWS also helped to identify missing requirements. We structured the newly identified requirements and revised the RD to accommodate them. These changes were also applied to the model. This shows that in order to achieve a more accurate requirement document and formal model the process of structuring requirements and modelling needs to be iterated.

We believe that the proposed approach can facilitate formal modelling of control systems and it can be used for modelling a structured RD using any refinement-based formal language. Furthermore, it is possible to use the MCC approach for organising requirement documents without modelling them formally.

Acknowledgments

The origins of the approach outlined in this section were based on experience with modelling the cruise control system of Bosch and Bosch provided valuable feedback on the approach. Southampton also undertook a collaboration with GM India Science Lab and researchers from GM provided valuable feedback on the approach.

3.3 Requirements Traceability

Tracing between informal requirements and formal models is challenging. A method for such tracing should permit to deal efficiently with changes to both the requirements and the model. A particular challenge is posed by the persisting interplay of formal and informal elements.

In this section, we describe an incremental approach to requirements validation and systems modelling. Formal modelling facilitates a high degree of automation: it serves for validation and traceability. The foundation for our approach are requirements that are structured according to the WRSPM reference model. We provide a system for traceability with a state-based formal method that supports refinement. We do not require all specification elements to be modelled formally and support incremental incorporation of new specification elements into the formal model. Refinement is used to deal with larger amounts of requirements in a structured way.

We provide a small example using Problem Frames and Event-B to demonstrate our approach.

3.3.1 Introduction

We describe an approach for incrementally building a formal model from structured informal specifications providing a means of requirements validation. Our approach does not require all specification elements to be modelled formally, and the resulting system description provides traceability to both formal and informal model elements. The traceability allows us to detect which requirements are affected if the system implementation changes, and vice versa. Most elements of the structured specification are still stated in natural language. Our aim is to increase the confidence that the formal model represents what has been specified, and to ensure that specification elements that do not have a formal representation are validated at a different stage of the development by informal reasoning and tracing. The ideas presented here have been developed successively in [JHLJ10, JG11a, JHL11].

We identified the WRSPM reference model [GJGZ00] as the foundation for the informal structured specification. Many concrete approaches are consistent with this reference model, e.g., [Jac01, PM95a]. A specification following the WRSPM approach can still be understood by stakeholders, while providing a good foundation for formalisation. These approaches define *phenomena* which describe the state space of the system and its environment, as well as artefacts that represent constraints on the state space and the state transitions. This structure makes a traceability to a state-based formalism doable.

A distinguishing feature of our approach is the incremental modelling of the specification using refinement, which the chosen formalism must support. Once modelled formally, the potential for automated verification is high. This is particularly useful for change management and requirements evolution, which are both important aspects for real-world systems. Also, we allow specification elements without formal representation. Those elements must be justified informally using techniques suggested in [Jac01], for instance.

3.3.2 Requirements and Specification

Our approach is based on WRSPM by Gunter et. al. [GJGZ00]. WRSPM is a reference model for applying formal methods to the development of user requirements and their reduction to a behavioural system specification.

WRSPM distinguishes between artefacts and phenomena. Phenomena describe the state space (and state transitions) of the domain and system, while artefacts represent constraints on the state space and the state transitions. The artefacts are broadly classified into groups that pertain mostly to the system versus those that pertain mostly to the environment. These are:

Domain Knowledge (W) describes how the world is expected to behave.

Requirements (R) describe how we would like the world to behave.

Specifications (S) bridge the world and the system.

Program (P) provides an implementation of S .

Programming Platform (M) provides an execution environment for P .

We distinguish phenomena by whether they are controlled by the system (belonging to set s) or the environment (belonging to set e). They are disjoint ($s \cap e = \emptyset$), while taken together, they represent all phenomena in the system ($s \cup e = \text{“all phenomena”}$). Furthermore, we distinguish them by visibility. Environmental phenomena may be visible to the system (belonging to e_v) or hidden from it (belonging to e_h). Correspondingly, system phenomena belonging to s_v are visible to the environment, while those belonging to s_h are hidden from it. These classes of phenomena are mutually disjoint.

The distinction between environment and system is an important one; omitting it can lead to misunderstandings during the development. It is sometimes regarded as a matter of taste or convenience where the boundary between environment and system lies, but it has a profound effect on the problem analysis. It clarifies responsibilities and interfaces between the

system and the world and between subsystems. If we require ourselves to explicitly make that distinction, we can avoid many problems at an early stage.

W and R may only be expressed using phenomena that are visible in the environment, which is $e \cup s_v$. Likewise, P and M may only be expressed using phenomena that are visible to the system, which is $s \cup e_v$. S has to be expressed using phenomena that are visible to both the system and the environment, which is $e_v \cup s_v$.

Once a system is modelled following WRSPM, a number of properties can be verified with regard to the model, one being *adequacy with respect to S*:

$$\text{FOR ALL } e \ s, \ W \text{ AND } S \text{ IMPLY } R \quad (\text{Adequacy})$$

This simply says that the specification constrains the world such that the requirements are realized. Obviously we are not interested in the trivial solution to (Adequacy), meaning that no e and s exist to satisfy (Adequacy).

We demonstrate our ideas using Problem Frames [Jac01], which is a concrete approach to software requirements analysis that is a manifestation of the WRSPM reference model. The central element in problem frames is the problem diagram that consists of exactly one machine domain, designed domains and given domains.

3.3.3 State-Based Modelling and Refinement

Our approach could be used with a wide range of formal methods for state-based modelling that have an associated notion of refinement. We find state-based formalisms such as ASM [BS03], VDM [Jon90], TLA+ [Lam02] or Event-B [Abr10] particularly suited because they permit straightforward specification of state, state invariants and state transitions for modelling dynamic behaviour. In this chapter, we focus on state-based modelling and provide an example using Event-B. Event-B is suitable for discussing the example that we introduce in Section 3.3.9. Using Event-B we can also discuss limitations of requirements tracing: not all requirements can be formalised within the core Event-B formalism. Formal and informal reasoning need to be combined in a sensible way. The boundary of formalisation in the example is given by temporal and real time properties. We have intentionally chosen a boundary that could be moved by using another formal method or extending Event-B because we think it is not fixed and may change depending on project characteristics. It also serves to illustrate that the boundary may be moved as a development progresses. We think of modelling and requirements validation as an incremental process: we permit the boundary to be moved

as need arises.

We take advantage of the concept of refinement which is supported by Event-B. Other notions of refinement could be used without changing the approach fundamentally. Our approach allows us to account for additional requirements at later refinement stages, thereby providing a structuring mechanism for the introduction of requirements into the formal model.

3.3.4 From Formal to Informal and Back

The requirements engineering process can be broken down into requirements specification, system modelling, requirements validation and requirements management [Wie03].

We will briefly describe these process steps (shown in Figure 3.11) and how they relate to the work in this chapter.

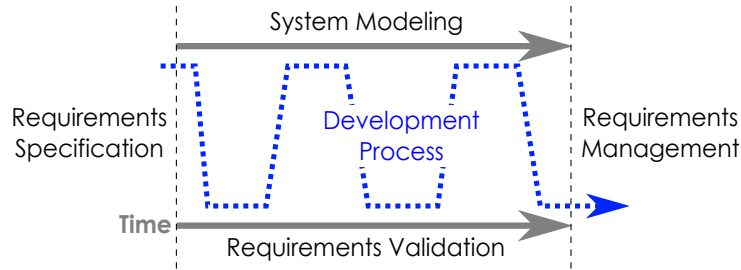


Figure 3.11: The Incremental Development Process

Requirements Specification – The requirements are structured according to the approach of choice, resulting in a specification that follows the WRSPM reference model. In the example that we introduce in Section 3.3.9, we use the Problem Frames approach.

System Modelling – The objective of this phase is the formal modelling of elements. Not all elements need to be modelled formally, which is one distinguishing feature of our approach. Also, this step can and should be performed incrementally. In general, any formalism can be used. The nature of the problem to be solved may suggest one formalism over another. In particular, it may be useful to select a formalism that makes it easy to model the safety critical aspects of the specification. It is also possible to use more than one formalism. In the example in Section 3.3.9, we use Event-B.

Requirements Validation – The validation of the requirements is a central aspect of this chapter and is described in detail in Section 3.3.5.

Requirements Management – In practice, a specification is never “done”.

The ongoing work includes change management and requirement evolution. These tasks are supported by our approach. The amount of formality determines how effective this is. At one end of the spectrum, all elements are modelled formally, allowing us to prove (Adequacy). On the other end of the spectrum is an informal description.

These tasks, including elicitation, analysis and negotiation, are performed in parallel. We do not want to create the impression that this is a sequential process.

3.3.5 Requirements Validation

System modelling provides us with partly formalised elements as described by the requirements. We think of system modelling as an incremental process where more and more is formalised. However, we do not assume that necessarily everything is formalised. The methodology we propose allows for a mixture of formal and informal proof as a means of validation. As a consequence of frequent incremental changes we need effective support for tracing requirements: formal models change as they incorporate increasing detail, requirements change as a consequence of the validation itself. The transition to requirements management is considered fluent and the same techniques of traceability are applied.

Demonstrating (Adequacy) now involves dealing with formal and informal elements. In the following, we designate by Rf the formal requirements, by Wf the formal domain properties and by Sf the formal specification elements. The difference $R \setminus Rf$ of all requirements and formal requirements gives the informal requirements Ri , similarly for informal domain properties Wi and informal specification elements Si .

For the formal elements we can formally verify that

$$\forall e s \cdot Wf \wedge Sf \Rightarrow Rf , \quad (3.1)$$

assuming that sufficient of W and S have been formalised to cover Rf . For informal elements we allow informal arguments, for instance, of the kind used in the problem frames approach [Jac01] or not formalised mathematical proofs. Doing this, we show:

$$\text{FOR ALL } e s, \ W \text{ AND } S \text{ IMPLY } Ri . \quad (3.2)$$

We permit also using formal elements in the antecedent of (3.2) but only formal elements in the antecedent of (3.1). As many critical requirements

as possible should be validated formally, giving high assurance of their satisfaction. Relying on formally verified facts in informal justification will also improve their quality.

Formal Tracing

To formalise artefacts A they need to be of a form that can be “translated” into a formula F so that we can state

$$A \text{ EQUIVALES } F . \quad (3.3)$$

This makes tracing from F to A and vice versa trivial. Formal proofs of (3.1) can provide information about which formal artefacts are used in order to validate specific requirements. Among others, this has been implemented in the proof support of the Rodin tool [ABH⁺10]. If formal artefacts $F_1, \dots, F_k$ have been used to prove formal requirement Rf_n from $Wf \wedge Sf$, then we know that a change of the informal requirement R_n that equivaless Rf_n affects the informal artefacts $A_1, \dots, A_k$. The formal model provides a way to validate requirements rigorously and an efficient way to trace dependencies between informal artefacts. The latter is crucial for the maintenance of large numbers of requirements occurring in industrial practice. Support by proof tools means that this tracing can be automated to a large degree.

Informal Tracing

Artefacts that are not formalised can still be traced but the dependencies can only be checked manually by inspecting informal arguments. Changes of involved artefacts require corresponding human intervention. A known technique to limit the impact of changes is the identification of a *satisfaction base* [KJ10] for each informal artefact of Ri . A satisfaction base for a requirement R_n consist of those artefacts from S and W that are sufficient to justify it. Using the concept of a satisfaction base, (3.2) can be rephrased as

$$\text{FOR ALL } e \text{ } s, \text{ } SB(R_n) \text{ IMPLY } R_n . \quad (3.4)$$

where $SB(R_n)$ is a subset of W and S , representing a satisfaction base for the given requirement. The satisfaction base is used in the informal justification and for tracing dependencies, similarly to formal tracing. However, possibilities for automation are very limited. Also note that there may be multiple satisfaction bases.

3.3.6 Formal Refinement

Formula (3.1) can grow very large for a complex model. This can make it very difficult to verify any interesting property but also to compute a sufficiently small set of formal artefacts that are used to verify specific formal requirements Rf_n . Formal refinement alleviates this problem by introducing parts of the overall model in small increments. The original WRSPM approach sketch a notion of implementation based on the program P and the programming platform M :

$$\text{FOR ALL } e \text{ } s, \text{ } W \text{ AND } P \text{ AND } M \text{ IMPLY } R. \quad (3.5)$$

This can be achieved by relying on implication for implementation (see, e.g., [HJ98, Heh93, GJGZ00]),

$$\text{FOR ALL } e_v \text{ } s, \text{ } P \text{ AND } M \text{ IMPLY } S \quad (3.6)$$

providing a simple notion of refinement in a predicative specification style. Instead of formalising the refinement notion (3.6) we prefer a notion based on discrete transition systems that permits more direct specification of dynamic aspects of a model. For the purposes of this article we do not consider details of M such as the targeted programming language. We consider S as a collection of invariants and transitions of a discrete transition system which we specify by means of Event-B [Abr10]. The choice of Event-B over similar methods [BS03, Jon90] is mostly motivated by the built-in formal refinement support and the availability of a tool [ABH⁺10] for experimentation with our approach.

3.3.7 Event-B Proof Obligations

In Event-B two main properties are proved about formal models: consistency and refinement.

Consistency. Consistency means that the invariant $I(v)$ is established by the initialisation

$$I(E_{\text{init}})$$

and maintained

$$I(v) \wedge G(t, v) \Rightarrow I(E(t, v))$$

by all other events of the machine. Usually, invariants are conjunctions, e.g., $I(v) = I_1(v) \wedge \dots \wedge I_n(v)$. Hence, it suffices to prove $I(v) \wedge G(t, v) \Rightarrow I_j(E(t, v))$ for all $j \in 1 \dots n$. The smaller predicates are easier to relate to informal artefacts and easier to trace in case artefacts correspond to theorems derived from the invariants.

Refinement. Refinement links abstract events to concrete events aiming at the preservation of properties of the abstract event when it is replaced by the concrete event. A concrete event with guard $H(u, w)$ and action $w := F(u, w)$ refines an abstract event with guard $G(t, v)$ and action $v := E(t, v)$ if, whenever the gluing invariant $J(v, w)$ is true:

- i. the guard of the concrete event is at least as strong as the guard of the abstract event, and
- ii. for every possible execution of the concrete event there is a corresponding execution of an abstract event which simulates the concrete event such that the gluing invariant remains true after execution of both events.

Formally,

$$I(v) \wedge J(v, w) \wedge H(u, w) \Rightarrow \exists t \cdot G(t, v) \wedge J(E(t, v), F(u, w)) .$$

For initialisation we have to prove: $J(E_{\text{init}}, F_{\text{init}})$. To match the refinement notion of WRSPM described in Section 3.3.6 we have to void data-refinement where a variable is replaced by another. We think data-refinement could eventually serve to deal with abstractions of phenomena where in more abstract problem frame descriptions phenomena are bundled. The Event-B method derives proof obligations from these two properties that are easier to handle and can be efficiently generated by a tool [Abr06]. In particular, the conclusion is decomposed into small parts. To achieve this witnesses $t = T(u, w)$ for t are introduced for instantiating the existentially bound identifiers:

$$I(v) \wedge J(v, w) \wedge H(u, w) \Rightarrow G(T(u, w), v) \wedge J(E(T(u, w), v), F(u, w)) .$$

Usually, guards and (gluing) invariants are conjunctions and the proof obligation can be decomposed similarly to the consistency proof obligation above.

3.3.8 Tracing of Requirements with Event-B

The Event-B model contains formal artefacts as indicated by (3.3). The domain properties Wf and specification elements Sf can be represented by means of events and invariants. By consistency and refinement we get a collection of invariants IA that are preserved by all events EA . We can now partition events and invariants according to the artefacts they represent: $IA = IW \cup IS$ and $EA = EW \cup ES$. Making this distinction is standard in the Event-B method.² To fit into the shape of WRSPM adequacy we consider the

²Using model decomposition we could now decompose the two parts and focus on the refinement of the sub-model consisting of IS and ES . The interface to the other sub-model would act as a contract guaranteeing overall consistency.

before-after predicates of all events and identify $Wf = IW \wedge BA(EW)$ and $Sf = IS \wedge BA(ES)$, where $BA(EW)$ yields the disjunction of the before-after predicates of the events EW . In formal refinement Pf the formal program is usually considered a subset of Sf that is being gradually constructed during refinement. After some refinement steps we have $Pf = IS \wedge IP \wedge BA(EP)$ where the events EP are refinements of the events ES . Hence, $Pf \Rightarrow Sf$ by choosing suitable witnesses, obtaining the formal counterpart of (3.6). We have identified the formal domain properties Wf , specification element Sf and program Pf .

We can now turn to the formal requirements Rf , formal adequacy (3.1) and the formalised (3.5) not taking account of the programming platform M :

$$\forall e \ s \cdot Wf \wedge Pf \Rightarrow Rf . \quad (3.7)$$

Assuming we already have verified (3.1), adequacy of the implementation (3.7) follows by the discussion of the preceding paragraph, using

$$\forall s \cdot Pf \Rightarrow Sf . \quad (3.8)$$

Refinement allows this to be applied incrementally to deal with small more manageable sets of artefacts at each refinement step. Gradually, the set of satisfied refinements is extended until all requirements are covered,

$$\emptyset = Rf^0 \subseteq Rf^1 \subseteq Rf^2 \subseteq \dots \subseteq Rf^n = Rf , \quad (3.9)$$

where the Rf^i correspond to the refinement steps of the model. Most of these refinement steps will involve the domain properties and specification elements:

$$\forall e \ s \cdot Wf^{i+1} \wedge Sf^{i+1} \Rightarrow Wf^i \wedge Sf^i . \quad (3.10)$$

Refinement steps for implementing the program will usually be less related to requirements. The refinement method, however, does not make a particular distinction between the two uses of refinement. Each refinement step can be used to verify adequacy of the specification gradually:

$$Wf^i \wedge Sf^i \Rightarrow Rf^i \setminus Rf^{i-1} . \quad (3.11)$$

Refinement theory guarantees that adequacy validated in earlier refinement steps is preserved. After n refinement steps (3.1) is verified.

Formula (3.11) suggests a method of stepwise tracing of requirements following the refinements. Often requirements can be identified with invariants, event guards or actions. In this case (3.11) holds trivially. Sometimes theorems can be stated [HL09] that are implied by the invariants. In this article

we limit tracing to this level. However, this is not a fundamental limitation of the approach. For instance, one could also permit temporal formulas derived from $Wf \wedge Sf$ as supported by TLA+. Some of TLA+ is also implemented in the ProB tool [LB08] that has been integrated with the Rodin tool that we use. But for this article we contend ourselves with a less expressive notation relying only on invariants and possible transitions.

Problem frame diagrams do not use refinement, but techniques of decomposition like projection. They serve for structuring large sets of requirements. They correspond to the last refined model just before turning to implementation (by means of P). The problem frame diagram will always contain the entire set of formal and informal requirements R . We do not intend to extend the idea of refinement from Event-B to problem frames in this chapter.

3.3.9 Example: A Traffic Light Controller

We are going to demonstrate the approach presented here by creating the model of a traffic light system that allows pedestrians to cross a street. We already introduced this example in [JHL11]. The system consists of two traffic lights for pedestrians (one on each side of the street), two corresponding traffic lights for the cars, and push buttons for the pedestrians to request a green light for crossing the street.

We consider this example useful, because it is simple enough to understand, but complex enough to be interesting. Further, the example concerns state (which we model formally) as well as real-time (which we specify informally), allowing us to demonstrate the mixing of formal and informal modelling elements. In the following, we only present the interesting aspects of the example.

3.3.10 Requirements Specification

Following our approach, we would apply a specification approach of choice, in this case Problem Frames. This may lead to a the problem diagram shown in Figure 3.12. The Problem Frames diagram is incomplete. For instance, information regarding the temporal properties of the system are missing. This is by design, as the problem diagram only depicts the contextual aspects of the model and their relationships in the form of shared phenomena. The textual representation is still the central repository for all information regarding the system. This leads to a new natural language specification, shown in Table 3.5. In the table, the phenomena are highlighted. The vocabulary is managed in a separate glossary (Table 3.6).

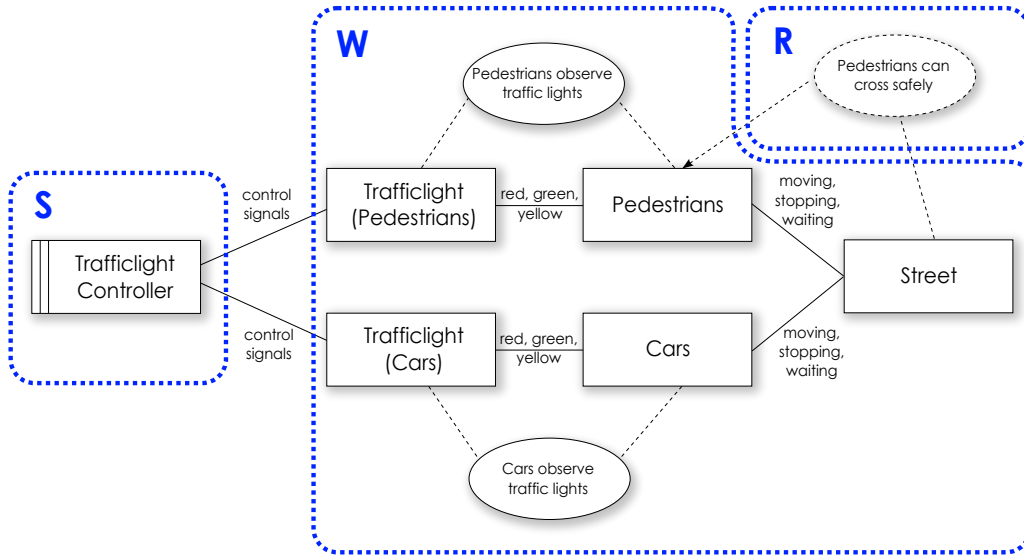


Figure 3.12: A simplified Problem Frames diagram for the traffic light problem

Note that it can be useful to introduce an informal notion of refinement already in the textual description of the system to structure it. We see that in the description of the traffic light states, that are sometimes referred to abstract as the abstract *stop* and *go*, and sometimes as the concrete colours *red*, *yellow* and *green*. We can take advantage of this in the modelling phase by establishing abstract properties that are simple and easy to trace. The refinement concept of Event-B allows us to introduce the concrete colours later on, while preserving the original properties (assuming correct data refinement), as demonstrated in Section 3.3.12.

The specification in Table 3.5 is already more precise than the original requirements, while still comprehensible by the stakeholders. We already identified items as *R*, *W* and *S*. This makes it easier to reason about the model. It also allows us to identify the proper role for validating or justifying each artefact: Stakeholders are concerned with *R*, domain experts with *W* and designers with *S*.

R-2.1 *Pedestrians* can cross safely. They are crossing when they are not *waiting*.

W-2.1	<i>Pedestrians</i> observe the traffic lights (tl_{peds}). This means that they may move (<i>moving</i>) when the traffic lights allows them to <i>go</i> . Upon indicating <i>stop</i> , they finish moving (<i>stopping</i>) and then wait (<i>waiting</i>).
W-2.2	<i>Cars</i> observe the traffic lights (tl_{cars}). This means that they may move (<i>moving</i>) when the traffic lights allows them to <i>go</i> . Upon indicating <i>stop</i> , they finish moving (<i>stopping</i>) and then wait (<i>waiting</i>).
W-2.3	<i>stopping</i> of <i>Pedestrians</i> takes time.
W-2.4	<i>stopping</i> of <i>Cars</i> takes time.
S-2.1	The traffic lights for pedestrians (tl_{peds}) and cars (tl_{cars}) never indicate <i>go</i> at the same time.
S-2.2	tl_{cars} must wait for a certain time ($delay_{cars}$) before switching to <i>go</i> after tl_{peds} turned to <i>stop</i> .
S-2.3	$delay_{peds}$ is 3 seconds ($\pm 100ms$).
S-2.4	tl_{peds} must wait for a certain time ($delay_{peds}$) before switching to <i>go</i> after tl_{cars} turned to <i>stop</i> .
S-2.5	$delay_{cars}$ is 3 seconds ($\pm 100ms$).

Table 3.5: Requirement, Domain Assumptions and Specification of a Traffic Light System (partial)

<i>Pedestrians</i> (e_h)	are modelled as <i>moving</i> , <i>stopping</i> or <i>waiting</i> .
<i>Cars</i> (e_h)	are modelled as <i>moving</i> , <i>stopping</i> or <i>waiting</i> .
tl_{peds} (s_v)	Traffic lights for pedestrians, modelled as <i>go</i> and <i>stop</i> .
tl_{cars} (s_v)	Traffic lights for cars, modelled as <i>go</i> and <i>stop</i> .
$delay_{peds}$ (e_v)	is modelled as an event that delays for 3 seconds after tl_{peds} turns from <i>go</i> to <i>stop</i> .
$delay_{cars}$ (e_v)	is modelled as an event that delays for 3 seconds after tl_{cars} turns from <i>go</i> to <i>stop</i> .

<i>go</i> (s_v)	is the state of a traffic light where only the <i>green</i> lamp is on.
<i>stop</i> (s_v)	are all states of a traffic light that are not <i>go</i> .

Table 3.6: The Glossary (partial)

3.3.11 System Modelling

We decided to use the Event-B formalism, making it easier to model some aspects of the model and more tricky to model others. In particular, it is easy to express safety properties like R-2.1, more difficult to express state transition properties like S-2.2, and almost impossible to express real-time properties like S-2.3.

Following the incremental approach described in Section 3.3.4, we start with the safety requirement R-2.1, for which a state-based formalism like Event-B is well-suited.

$$Pedestrians \neq waiting \Rightarrow Cars = waiting \quad (3.12)$$

Not all properties can be modelled as easily as R-2.1. For instance, the behaviour of pedestrians (W-2.1) cannot be represented by an invariant. Instead, we can model it according to the approach described in Section 3.3.8 by representing it as a before-after predicate of an event. The property W-2.1 doesn't have the proper granularity for this approach, so we rewrite it to specify each transition separately. This rewrite is part of the incremental specification process, and the result must be validated with the domain experts.

W-2.1 (a) *Pedestrians* that are *moving* can only change their state to *stopping*.

W-2.1 (b) *Pedestrians* that are *stopping* can only change their state to *waiting*.

W-2.1 (c) *Pedestrians* that are *waiting* can only change their state to *moving*.

W-2.1 (d) *Pedestrians* may only change to *moving* if tl_{peds} indicates *go*.

W-2.1 (e) If tl_{peds} indicates *stop*, then *Pedestrians* must change to *stopping* if they are *moving* and change to *waiting* if they are *stopping*.

Rewritten like this, it can be modelled in Event-B as follows:

```

Event  peds_moving_to_stopping  $\hat{=}$ 
    when
        W-2.1a : Pedestrians = moving
    then
        W-2.1a : Pedestrians := stopping
Event  peds_stopping_to_waiting  $\hat{=}$ 
    when
        W-2.1b : Pedestrians = stopping
    then
        W-2.1b : Pedestrians := waiting
Event  peds_waiting_to_moving  $\hat{=}$ 
    when
        W-2.1c : Pedestrians = waiting
        W-2.1d : tl_peds = go
    then
        W-2.1c : Pedestrians := moving

```

Note how we could establish a clear traceability according to (3.3). The exception is W-2.1e, which is difficult to model in Event-B. Event-B allows us to enforce that something does *not* happen (via a guard), but difficult to guarantee that something does happen (implying that all events except one are disabled). The missing traceability to W-2.1e reminds us that this property must be justified outside this formal model. This could be done by reasoning, testing, or with a different formalism like temporal logic.

This justification may be invalidated if the source or target of the traceability relationship changes. Thus, it has to be verified after each such change. A tool may support this by invalidating that relationship if either of the elements involved changes.

The reader may have noticed that the above represents a state machine. It could be useful to develop an approach specific to state machines.

3.3.12 Data Refinement

In Section 3.3.7, we described how consistency is maintained across refinement levels. We will demonstrate this concept by showing how the traffic light states *stop* and *go* are transformed via data refinement into *red*, *yellow* and *green*.

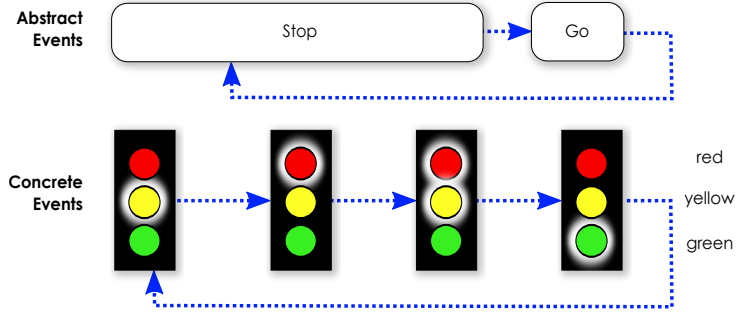


Figure 3.13: Data Refinement of the Traffic Light States

Data refinement allows us to state abstract properties in a concise way, while the implementation details are addressed later. This allows us to reason about some fundamental properties. Consider S-2.1 as an example of such a property. By arguing simply about *stop* and *go*, the safety property can be stated in a very concise way. The detail on how *stop* and *go* are realised (through colours), can be provided later. Carrying the notion of refinement to the requirements allows us to write more concise requirements: In this case, we can separate the safety requirement from the actual representation of traffic light states, which is also a requirement, but a different one.

There are other situations where this approach can be exploited: For product lines, some abstract properties could be realised in different concrete implementations. In this example, *stop* and *go* could be signalled with a barrier, as found in railroad crossings. A carefully crafted abstraction would therefore support the automated verification of different concrete implementations.

We can model S-2.1 formally as follows:

$$\neg (tl_{peds} = go \wedge tl_{cars} = go) \quad (3.13)$$

The definition of *stop* and *go* in terms of colours was already given in Table 3.6, leading to the following gluing invariant that can be introduced in a new refinement:

$$tl_{peds} = go \Leftrightarrow colors_{peds} = \{green\} \quad (3.14)$$

Introducing (3.14) into the model results in non-discharged proof obligations, as the newly introduced gluing invariant will be violated without any further modifications. The abstract events that control the traffic light's *stop* and *go* states must also be refined into concrete events that cycle through the corresponding colour states, as shown in Figure 3.13.

The refinement will take on a similar form as the Event-B shown in Section 3.3.11, where each state transition corresponds to one event. The proof obligations will ensure that the safety requirement (3.13) is not violated once they are all discharged, assuming that the gluing invariant (3.14) is modelled correctly. Discharging all proof obligations will require additional guards.

3.3.13 Adding Requirements with Refinement

Another application of refinement is the gradual inclusion of formal requirements into subsequent refinements, as hinted at in (3.9). In the traffic light example, this can be demonstrated by adding a push button for the pedestrians, allowing them to request crossing the street.

Table 3.7 shows the structured requirements and their formal representations:

R-2.2	<i>Pedestrians</i> can <i>request</i> to cross any time.
S-2.6	Upon switching of <i>tl_peds</i> from <i>go</i> to <i>stop</i> , the <i>request</i> is reset.
S-2.7	<i>Pedestrians</i> must not wait longer than 60 seconds for permission to cross after issuing the <i>request</i> .

Table 3.7: Requirement and Specification for allowing Pedestrians to Request Crossing the Street

These two properties can be incorporated into the model in a separate refinement with a new event and the extension of an existing event with a straightforward traceability, as shown in the following:

```

Event  request_crossing  $\hat{=}$ 
    when
        R – 2.2 : request := TRUE
    end

Event  set_tl_peds_go  $\hat{=}$ 
extends set_tl_peds_go
    when
        S – 2.6 : request := FALSE

```

The requirement S-2.7 cannot be modelled formally as stated. This informal artefact simply has to be verified outside the formal model. We could

break down S-2.7 further to model some aspects formally (e.g. by introducing a “tick” interval). Our approach could handle this, but we omitted this for brevity.

, but there are techniques that In this case we decided to not model S-2.7 formally, something that our approach can handle without difficulty.

3.3.14 Tool Support

A tool supporting this approach would have to provide a mechanism to mark informal artefacts to be marked as “justified”, and a place to write this justification down. Further, all R_i would have to be marked for re-justification, as soon as any W or S changes.

We developed a platform for requirements engineering called ProR³ [JG11a]. While the tool can be used stand-alone, we designed it with the goal to ease the integration of natural language requirements and formal models.

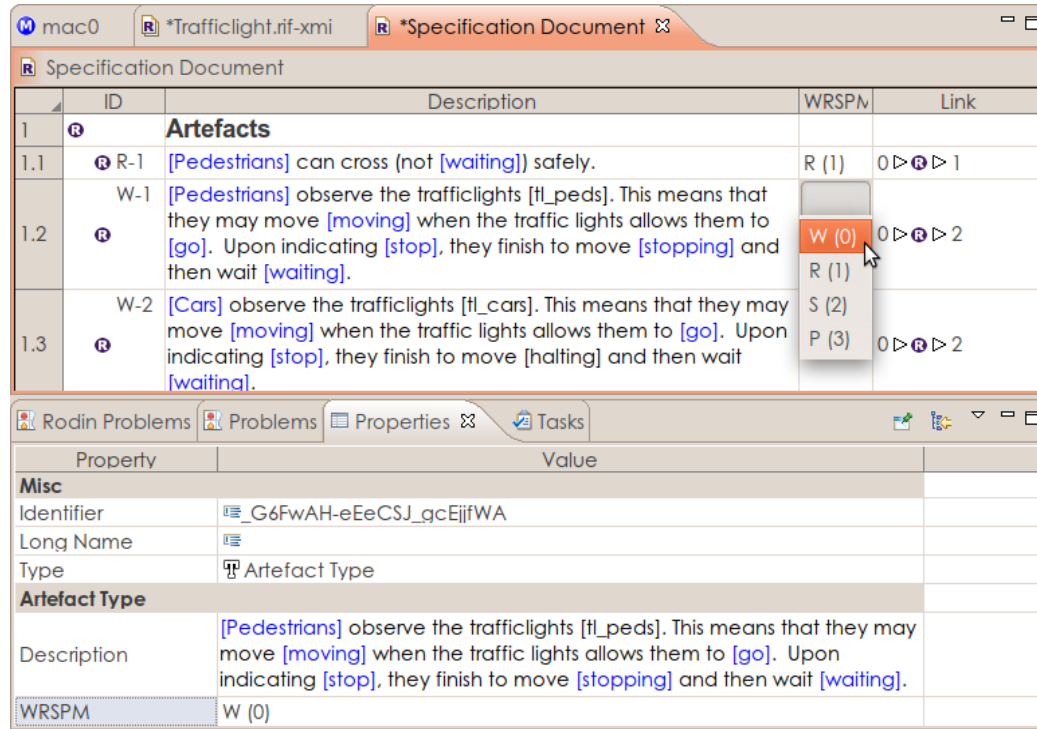


Figure 3.14: Integration of WRSPM-structured artefacts and formal Event-B elements.

ProR is based on the Requirements Interchange Format RIF/ReqIF [OMG11].

³<http://www.pror.org>

RIF was created in 2004 by the “Herstellerinitiative Software”, a body of the German automotive industry that oversees vendor-independent collaboration. In 2010, the Object Management Group (OMG) took over the standardisation process and released a new version of the standard under the name ReqIF. Our tool environment is currently based on RIF 1.2, support for ReqIF 1.0 is planned.

ProR is part of the Requirements Modeling Framework⁴ (RMF), which is an official Eclipse Project.

ProR can be installed directly into Rodin. A tight integration can be achieved with plugins that access both the Rodin and ProR data structures.

We created a plugin that allows us to manage the vocabulary of the natural language requirements as Event-B models. Via this plugin, ProR supports highlighting of formal model elements directly in the requirements text. Annotated traces can be used to record information regarding relationships. For instance, this mechanism can be used to record the justification argument between a textual requirement and a formal model element.

Formal Event-B elements have a corresponding proxy object in the RIF model that is automatically synchronised with the Event-B model. The integration is currently manual via drag and drop. The proxy object can be extended with additional attributes to store arbitrary information.

The plugin is built using the Eclipse EMF technology⁵. This allows us to “hook” code into the models to perform various tasks. Depending on the specification approach used, we could provide validators to ensure consistency according to the approach taken.

The application of the tool is shown in Figure 3.14, where the elements from the formal model are highlighted in the requirement text. We also see how a classification of elements can be performed, in this example following WRSPM. The desired artefact type is selected from a drop-down directly in the editor.

The *Properties View* in the lower pane shows additional information regarding the selected element.

The right column shows the number of incoming and outgoing links, providing a quick summary of each element’s traceability. These links can be unveiled, as shown in Figure 3.15. Rows with a triangle represent an annotated trace. In this example, an informal justification has been provided.

For links, the rightmost column contains the link target. Selecting it shows the target’s properties in the Property View. In the screenshot we see that the link target is the event *stopping_peds*. As it is selected, the Property

⁴<http://eclipse.org/rmf>

⁵Eclipse Modelling Framework, <http://www.eclipse.org/emf/>

View shows its attributes, including the event itself. This is a reference to the model, not a copy of the event.

The tool is currently in a prototypical state and is actively developed. Specifically, it currently support the manual creation of links and colour highlighting. We envision a tool that identifies unaccounted requirements and model elements, and that invalidates traces when related model elements change, as well as change impact analysis.

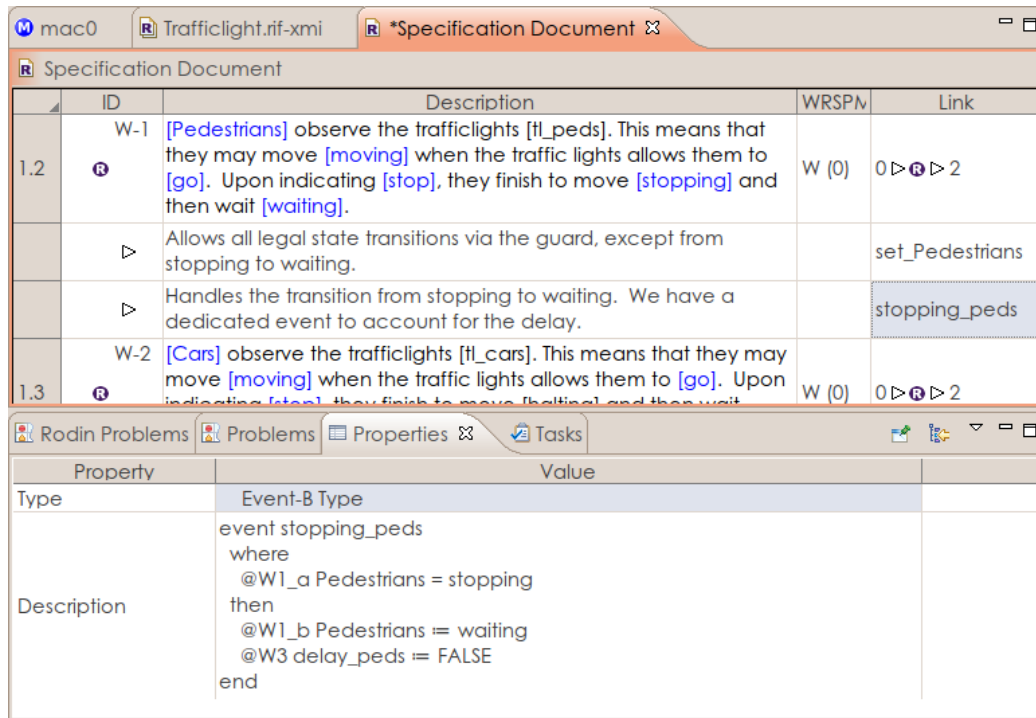


Figure 3.15: The unveiled traces of an element. As the link target is selected, the link target’s properties are shown in the Property View (the lower pane)

3.3.15 Related Work

The ideas of the approach presented in this chapter have been developed successively in [JHLJ10, JG11a, JHL11]. The tool platform ProR has been presented before in [Jas10, JG11a, JG11b].

The issue of traceability has been analysed in depth by Gotel et. al. [GF94]. Our research falls into the area of post-requirements specification traceability.

Abrial [Abr06] recognises the problem of the transition from informal user requirements to a formal specification. He suggests to construct a formal model for the user requirements, but acknowledges that such a model would

still require informal requirements to get started. He covers this approach in [Abr10].

The WRSPM reference model [GJGZ00] was attractive because it allowed us to discuss the specification in general terms, while still being meaningful in the context of a specific approach like Problem Frames [Jac01] or the functional-documentation model [PM95a].

There have been successful attempts in applying Problem Frames and Event-B together. In [LGG⁺10], the authors show how these are being applied to an industrial case study. In contrast to our approach, only requirements that were actually modelled formally were included in the specification in the first place.

There are approaches spanning from requirements to formal model, a well-known one being KAOS [DDMvL97]. But rather than allowing informal elements that are omitted from the formal model, it provides so-called “soft-goals” that are broken down into requirements that can still be modelled formally.

Reveal [Pra03] is an engineering method based on Michael Jackson’s “World and the Machine” model. There are a lot of similarities to our approach, including the acknowledgement of requirements that are not part of the formal model. However, Reveal is more of a process description of the overall requirements engineering process. Therefore it could be quite attractive to apply the Reveal process with the approach described here.

Last, [WAC10] describes a much more comprehensive case study where a number of the concepts described in this chapter can be found.

3.3.16 Conclusion

In this section, we presented an approach for incrementally building a formal model from structured informal requirements. Our approach supports partial formal modelling and provides traceability for both formal and informal specification elements. This approach allows us to take advantage of the formal model regarding automated verification, while providing a systematic (albeit manual) approach to validation of the remaining specification elements.

We demonstrate our ideas on a specification and model of a traffic light system. While this is arguably a teaching example, it contains examples of specification elements that are challenging in formal modelling and demonstrates how these can be addressed.

We believe that tool support is a crucial element for such an approach to work and presented an integration of the ProR platform for requirements engineering and the Rodin platform for Event-B modelling to support our approach.

Future Work. We will continue investigating different specification methods. While we find WRSPM useful, it is a reference framework that is not intended to be applied as is. We have experimented with Problem Frames, which are useful but does not match well with our approach to refinement (based on Event-B).

We will explore the suitability of Event-B for modelling bigger specifications with our approach, if possible real-world examples.

Last, we will continue our work on tool support.

Chapter 4

Model Animation

J. Bendisposto, S. Hallerstede, M. Jastram, M. Leuschel, D. Plagge

4.1 Refinement Animation

4.1.1 Introduction

The Rodin tool generates automatically proof obligations that can be analysed to gain insight into a formal model. They often give good indications of how to make an improvement in case of inconsistencies in a model. However, there are also many occasions where proof obligations do not point directly to a problem or where a model does not contain inconsistencies but is still “incorrect” (see, e.g., the Earley parser example discussed in [BLLS08]). In many such cases animation is an useful tool to gain further insight into a model.

PROB was originally written for the B method and had a very rudimentary support for animating refinements. We present here how we improved PROB to animate refinements in Event-B. This should help users to locate the cause of problems in the model, pointing to specific invariants, guards, and so on, if possible.

The article is based on the work in [HLP11], where we also give a concise description about the fundamentals of Event-B and the motivation behind its concepts. Here we assume that the reader already has a basic understanding of Event-B’s concepts.

4.1.2 Animating Event-B with Refinements

Refinements are already a central concept in the B method. But PROB has only limited the support for animating refinements. Event-B supports in

contrast to classical B witnesses that provide a gluing between the abstract machine and the refinement.

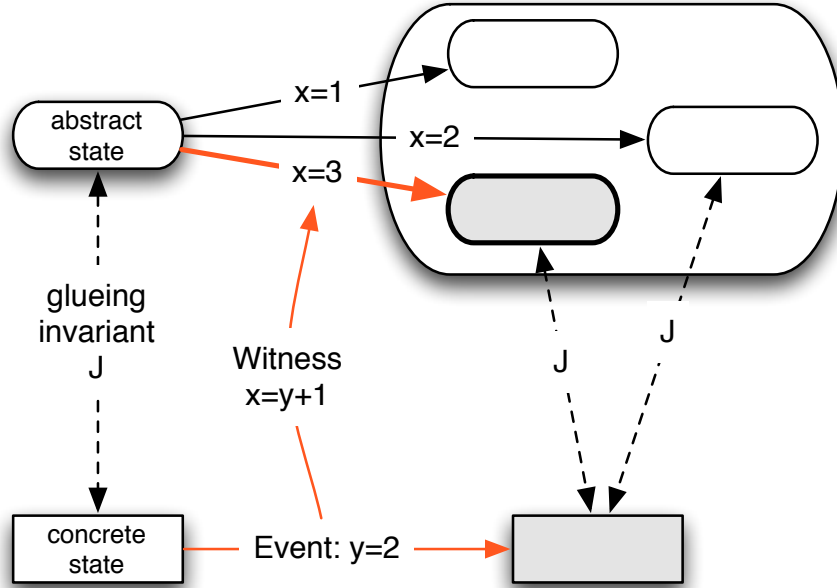


Figure 4.1: Witnesses

Witnesses do not only simplify proof, they also help to link concrete and abstract values in a refinement as seen in Figure 4.1. For a refined operation in classical B it must be proven that for each possible execution of the concrete event there is an corresponding execution of the abstract event such that the invariant holds afterwards. When trying to animate such machines, the computation of possible execution paths can become very complex because there is no explicit link between values of concrete and abstract variables and the invariant has to be taken into account for calculating the operation's effect. Thus witnesses play a central role when animating models with complex refinement relations because often they pinpoint from values of refined events to values of abstract events.

4.1.3 Contexts

In a context, new carrier sets and constants can be defined and statements about them can be expressed by axioms. Contexts can be animated, too. In that case, PROB searches for values of the constants that fulfil all axioms. Possible solutions are shown to the user as if setting the constants would

be an event called *SETUP_CONTEXT*. If a machine that sees a context is animated, first the context is set up before the user can continue with the initialisation and then with the other events.

```

context  CofCtxt

constants  full  empty  half  level

sets  FILL

axioms

  @Ffhe  partition(FILL, {full}, {half}, {empty})

  @lv1  level = (0 .. 2 × {empty}) ∪ (3 .. 7 × {half}) ∪ (8 .. 11 × {full})

end

```

Figure 4.2: Defining the constants in a context of coffee dispenser model

4.1.4 Running Example

We use the coffee dispenser model in Figure 4.3 (with the context shown in Figure 4.2). and Figure 4.5 for illustration of refinement-animation. In the abstract machine *CoffeeM* the dispenser can fill a mug half or fully; the state of the mug is represented by the variable *alvl* (abstract level). As a special service the dispenser can also drink the coffee.

In the first refined machine *CoffeeR1* a feature is introduced for inserting an arbitrary number of coins into the dispenser. A coin is consumed each time a mug is filled. In the second refined machine *CoffeeR2*, the number of coins maximally accepted is limited and the amount of coffee contained in a mug is represented numerically by a the variable *clvl* (concrete level).

4.1.5 Animation in ProB

The before-after predicate can be used to compute the state space of a machine, a graph where each node represents a state of the machine and each arc the execution of an event. In Figure 4.3 you can see on the right side the state space of the machine *CoffeeM* (on the left side), as computed by the ProB tool. The triangle represent a special root node, where the variables and constants of a machine have not yet been set. An animator lets the user navigate the state space by choosing the events to be fired. A model checker

will systematically explore the state space, looking for various errors in the machine.

Figure 4.4 shows a screenshot of PROB in action. In the upper left corner the events are listed and the user can select one to fire it. Below of events is a list of recently fired events and on the right the values of current variables is shown.

Inspection of a state

PROB provides an interface that let's the user inspect a state. The values of all variables are shown. Additionally, the user can inspect the axioms and invariants, i.e. he can see whether they are true or false, and sub-expressions can be shown- The guards of the events are shown in a similar manner. The user can also enter new predicates or expressions that are evaluated and shown for each state.

Validation of the machine's consistency

The consistency of a machine is proven by showing that the initialisation establishes the invariant and that each event change For each state PROB checks if the invariant holds. If not, an error is reported. Another proof obligation that belongs to machine consistency is action feasibility, i.e. non-deterministic actions must be feasible if the guard is true. PROB checks that this holds, too.

Additionally, in each state is checked if the theorems hold.

Another feature of PROB is the “constraint based invariant check”. Instead of traversing the state space of the model, PROB tries to find values for all variables before and after an event that fulfil the following properties:

- The invariant must be true for the values before the event occurs.
- The invariant must be false for the values after the event occurs.
- The event's before-after-predicate must hold.

With this approach, PROB can find counter-examples that cannot be found by exploring the state-space. On the downside, sometimes finding suitable values takes too long for the user to wait for.

Enabledness

Currently Event-B does not provide tool support for proving liveness properties. PROB is able to check for deadlocks, i.e. for states where no event

is enabled. This can be done by exploring the state space or by using the “constraint based deadlock check” that works analogously to the constraint based invariant check.

4.1.6 Refinement validation

When animating refinements, PROB keeps track of all variables of all refinements. The algorithm (see Section 4.1.8) takes a concrete event and then computes the refined abstract events successively. We now give a brief overview over the properties that a refinement must have and that are checked by PROB:

Guard Strengthening A concrete event must only be enabled when the abstract event is enabled. PROB evaluates the abstract guard if the concrete guard is true. Witnesses provide values for abstract parameters.

Witness Feasibility If the abstract event has a parameter that is dropped in the concrete event or it has a non-deterministic assignment, a witness for the abstract parameter or the new value of the abstract variable must be given. PROB uses the witness to compute the values. If no value can be found, it has found a problem with witness feasibility.

Action Simulation The abstract actions should be able to assign the same values as the concrete event or those given by witnesses. If PROB does not find such values, it has found a counter-example for the action simulation proof obligation.

Variable preservation If a concrete event assigns a value to a variable that is not changed in the abstract event, PROB checks if the value is the same as before the event occurred.

4.1.7 Termination

In Event-B events can be marked as “anticipated” or “convergent”. In this case such events must not increase resp. must decrease a variant. PROB can check this easily by comparing the old and new state for each computed event.

4.1.8 The Animation Algorithm

In this section we describe the validation and animation algorithm in detail. We point out in the presentation of the algorithm how it indicates problems

with particular proof obligations. We also show how feedback to the user needs to be considered. Producing informative output from an animation with good performance is a challenge. For this reason the algorithm makes heavy use of PROB's existing functionality. In particular, PROB provides methods to find values for variables that satisfy predicates occurring in Event-B models.

We limit the discussion to animation. For model checking, the animation algorithm can be used to automatically explore the state space of the model.

4.1.9 Preprocessing

The algorithm is applied to a particular refinement machine M_i of a model. In a pre-processing step, all ancestor machines $M_0, \dots, M_{i-1}$ of M_i are loaded and all contexts seen by $M_0, \dots, M_i$ are merged by collecting the declared constants and joining the axioms. The invariant is obtained by conjoining all invariants of $M_0, \dots, M_i$.

We transform each event of M_i to an internal representation. The representation is outlined on the right hand side of Fig. 4.6. Usually the list of abstract events contains just one entry. If an event refines *skip* or belongs to the most abstract machine M_0 , the list of abstract events is empty. If the event refines several events, it will contain all of those events.

E.g., if we animate *CoffeeR1*, the event *drink* has one abstract event, the event *drink* of *CoffeeM*, which itself has an empty list of abstract events. *insert_coin* would have no abstract events because it refines *skip*.

4.1.10 The Animation Algorithm

The animator executes events depending on the current state of a model. It maintains a state consisting of all constants of the seen contexts as well as all variables of the machines $M_0, \dots, M_i$.

In a first step the animator tries to find values for the constants that satisfy all axioms. Subsequently, the animator executes in each step an event of the most concrete machine M_i ; then it executes the corresponding abstract events from the concrete event to the most abstract event.

When all of this has been done, the animator is ready for the next step.

The algorithm to animate a particular event works as follows (item numbers correspond to those of Fig. 4.6, left hand side):¹

¹With respect to animation *INITIALISATION* is not treated differently from any other events (see left of Fig. 4.6) except that it is enforced to occur once upon start of an animation.

1. Search for possible values for the parameters by evaluating its guard. If no values are found, the event is disabled.
2. Execute each action by evaluating the respective before-after predicate. If no solution is found, report an error. The possible reasons for a failing action are:
 - (a) A violation of the event feasibility proof obligation **FIS** is found if the predicate P of an action $v :| P$ is not satisfiable or the set S of an action $v : \in S$ is empty.
 - (b) A violation of the action simulation proof obligation **SIM** is indicated if the new value v' of a variable v is constrained by a witness of a refined event (see step 4), but the abstract action cannot assign a value to v' such that the witness holds.
3. If a new value v' is different from its old value v and v is an abstract variable that is not changed by the refined event, a violation of the variable preservation proof obligation **EQL** is found.
4. For each witness evaluate its predicate and try to find values for the witnessed variable. If no value is found for a witness, report an error, because a witness should have at least one solution (by the witness feasibility proof obligation **WFIS**).
5.
 - (a) If the list of abstract events is empty, a complete solution has been found for this event that leads to a new state. It consists of the values newly assigned by the actions plus the variables unchanged by the actions.
 - (b) If there are one or more abstract events, choose one non-deterministically and evaluate its guard like in step 1. If it evaluates to true, continue recursively with step 2, otherwise try the next event.
If no guard evaluates to true, report an error, because the guard of the refinement is weaker than that of the abstract event (violation of the guard strengthening proof obligation **GRD**).

All four steps can be non-deterministic and we generate all solutions (limited to a maximum number) with backtracking.

Common Variables and Parameters

As mentioned above we have basically two options for dealing with common variables between a model and its refinement. One is to treat them as distinct

variables x_a and x_c and to add the implicit gluing invariant $x_a = x_c$. If an abstract event $x := 1$ is now erroneously refined by $x := 2$, the animator can visualise the error by showing the post state $x_a = 1, x_c = 2$ and indicating that the invariant $x_a = x_c$ is violated.

But this approach has a fundamental problem as the following example shows. Let the action of the abstract event be $x \in \{1, 2\}$ and the corresponding action of the refined event $x := 1$. Our algorithm would find two possible post states, $x_a = 1, x_c = 1$ and $x_a = 1, x_c = 2$. The first one is valid, the second one violates the implicit gluing invariant. Thus PROB would wrongly show an error for a correct refinement.

A solution to prevent the spurious error is to add implicit witnesses $x'_a = x'_c$ for all common variables. Then after assigning $x'_c = 1$, the evaluation of $x'_a = x'_c$ results in $x'_a = 1$. The abstract assignment $x \in \{1, 2\}$ is then just a check $x'_a = 1 \in \{1, 2\}$. We still would find an error for an invalid concrete action (like $x := 3$), namely a violation of the action simulation proof obligation in step 2a. But adding the implicit witness has the effect that the algorithm never finds solutions with different abstract and concrete values. So there is no need to distinguish between them in the animation. The same applies to common parameters.

A slightly different solution would be to add implicit witnesses only for those variables which are assigned non-deterministically in the abstract event. This would allow us to generate a state which violates the gluing invariant that we could present the user in case of deterministic assignments. For non-deterministic assignments we still have to indicate a violation of the action simulation proof obligation in the event. It is not clear to us if the presentation of an invalid result state is easier to understand than the explanation of an error found in the event. So there is no clear benefit of this approach, but the downside would be a less consistent algorithm and a larger representation of a state by duplicating variables. Thus we have chosen the approach where we do not distinguish between common variables.

Example

Fig. 4.7 shows a detailed example run for the event *fill_mug* in the refinement *CoffeeR2* to demonstrate the animation algorithm. The table displays what is happening for each refinement level and step of the algorithm. The last column contains an entry if new values for variables have been found by evaluating the predicates.

We have chosen an arbitrary state where *fill_mug* is enabled as a starting point: $alvl = \text{empty}$, $clvl = 0$, $coins = 2$, $maxc = 4$. The example shown leads to a new state $alvl = \text{full}$, $clvl = 9$, $coins = 1$, $maxc = 4$. The variable

maxc has not been modified by any action, so its value is just copied to the new state.

Note that the evaluation of action @ffl (*CoffeeR2*, step 2) chooses the value 9 of *clvl'* non-deterministically, in total there are 4 different possibilities, leading to 4 different states with $clvl \in 8 \dots 11$.

Animating Only a Part of the Refinement Chain

Above we presumed that the user wants to animate a refinement M_i and all its ancestors $M_0, \dots, M_{i-1}$. But we also permit the user to limit the animation to the refinements between M_i and an “upper” refinement M_k with $0 \leq k \leq i$ instead of M_0 . Then variables of not animated models and parts of predicates that contain references to those variables will be removed, resulting in weaker predicates.

Such a reduction of the animated refinement levels usually leads to smaller representation of states and allows the user to focus on the parts of the model he is currently interested in. However, the downside is that the limitation might hide some errors or introduce spurious errors:

- The gluing invariant can refer to abstract variables, weakening it means that some invariant violations are not detected anymore.
- Weakened witnesses can result in finding values for abstract parameters or variables that were not possible with the original predicates. Those values might cause other errors (e.g. violation of the simulation proof obligation).

If we limit the animation to a single refinement ($k = i$) the animation behaves like that of PROB without multi-level support.

4.1.11 Empirical Evaluation

Symmetry Reduction for Multi-Level Validation

The reason symmetry reduction cannot be (easily) applied when using the refinement checking algorithm from [LB05], is that the abstract and refined model are explored separately. Hence, symmetry reduction would have to be applied to these two steps separately as well. A big problem now is that the canonical form chosen for the abstract model may be incompatible with the one chosen for the refined model, leading to erroneous counterexamples and/or to real counterexamples not being found.

In multilevel animation, the canonical form is computed on the combined state of the abstract and refined model, and as such respects the relationship

Cardinality of Proc	States (without symmetry)	Validation Time	States (with symmetry)	Validation Time
1	3	0.001 s	3	0.002 s
2	6	0.004 s	4	0.003 s
3	17	0.012 s	5	0.006 s
4	66	0.049 s	6	0.009 s
5	327	0.278 s	7	0.013 s
6	1,958	1.618 s	8	0.018 s
7	13,701	11.325 s	9	0.025 s
8	100,001	90.930 s	10	0.033 s

Table 4.1: Experimental results for multi-level validation with symmetry reduction

between them. This is illustrated in Fig. 4.9, where one representative is chosen for the combined abstract and refined state.

In Table 4.1 we show how effective symmetry reduction can be for multi-level validation. The model of *ProcSeq* can be found in [HLP11]. We have run exhaustive multi-level model checking for the refined *ProcSeq* model of with and without symmetry reduction. We have used the symmetry reduction technique from [SL08].

More Experimental Results

The experiments below were again run on a MacBook Pro with a 3.06 GHz Core2Duo processor, and PROB 1.3.2 compiled with SICStus Prolog 4.1.2.

To evaluate the cost of treating multiple levels at once we have run our algorithm on an Event-B model for an elevator by ETH Zürich with an abstract machine and 13 refinements. We have measured the time to check 100 states for each of the 14 machines. The results are summarised in Table 4.2. Note that the state spaces for levels 0 to 4 have less than 100 nodes each, hence the full state space was explored in the given time. Starting at level 5, the state space has more than 100 states; we have stopped the algorithm at 100 nodes to be able to compare the time required to treat the same number of nodes. Also note that, we have always animated all levels above the given level (i.e., when validating the level 13 model, we have also validated levels

Level	Validation Time (100 states)	Cost for extra level	Time per level
0	0 ms		
1	0 ms		
2	20 ms		
3	30 ms		
4	80 ms		
5	100 ms		
6	130 ms	30.0%	18.57
7	160 ms	23.1%	20.00
8	190 ms	18.8%	21.11
9	840 ms	342.1%	84.00
10	1110 ms	32.1%	100.91
11	1550 ms	39.6%	129.17
12	1790 ms	15.5%	137.69
13	2160 ms	20.7%	154.29

Table 4.2: ETH elevator (levels 0–4 have less than 100 states)

0–12). As we can see, there is no simple linear dependence between the validation time and the number of levels animated. However, up to level 8 the runtime seems to increase linearly with the number of levels animated. Then there is a big jump at level 9 (which introduces 6 new variables of type total function), after which the runtime seems to increase again linearly (albeit with a steeper slope).

We have conducted a similar experiment for the Quicksort model from [Hal09a], for a particular array to be sorted. The results are summarised in Table 4.3. Here, we can see that between level 0 and 1 the number of nodes and the validation time actually decreases: the refined model is more deterministic and has, hence, a reduced state space. The non-deterministic choice

Level	Validation Time (complete)	Nodes	Cost for extra level	Time per node and level
0	300 ms	34		8.82
1	60 ms	18	-80.0%	1.67
2	110 ms	18	83.3%	2.04
3	170 ms	44	54.5%	0.97
4	190 ms	65	11.8%	0.58
5	260 ms	65	36.8%	0.67
6	310 ms	66	19.2%	0.67
7	360 ms	66	16.1%	0.68
8	410 ms	66	13.9%	0.69
9	460 ms	66	12.2%	0.70

Table 4.3: Quicksort model

of a sorting permutation in the abstraction is implemented by a deterministic algorithm computing such a permutation. The validation time per node and level actually goes down until level 4, and then remains relatively stable from level 5 onwards.

We have also applied our algorithm on industrial models from SAP, Bosch and Siemens within the Deploy project. In all instances, the multi-level algorithm dealt very well with large number of refinement levels.

4.1.12 Related Work

Comparison with Trace Refinement Checking

An important related work is the refinement checking algorithm in [LB05]. This algorithm checks trace refinement between the abstract model and the refined model, i.e., it checks that every trace of events of the refined model is also a valid trace of the abstract model.

For example, for our running *Coffee* example, all the possible traces of the abstract model can be deduced from the state space in Figure 4.3. If the refined model can perform the trace *INITIALISATION*, *drink*, then the algorithm from [LB05] would detect an error, as this trace cannot be performed by the abstract model *CoffeeM*.

The main differences between our new multi-level validation algorithm and [LB05] can be summarised as follows:

- As the algorithm from [LB05] works on the level of traces, it can be applied to other formalisms, such as CSP [Hoa85]. It can even be used to check refinement relations between CSP and classical B [Abr96] models.
- The tools for classical B and Event-B generate proof obligations for forward refinement, which is not complete. I.e., there are systems which are related by trace refinement but where no forward refinement can be established [HHS86]. The algorithm in this chapter adheres to the forward refinement methodology of Event-B, whereas [LB05] checks trace refinement.
- The algorithm of [LB05] checks the refinement relation between two models; our new algorithm checks an entire refinement chain in one go (but restricted by the state space of the most concrete model).
- The algorithm of [LB05] computes the state space of the refined model on-the-fly, but it has to compute the entire state space of the abstract

model *before* starting the refinement checking (for the running *Coffee* example, this corresponds to Figure 4.3). The algorithm in this chapter performs the entire validation on-the-fly. In particular, only the relevant parts of the state space of the abstract model(s) are computed. We will illustrate this in the empirical results below.

- The algorithm of [LB05] performs the refinement checking depth-first; the new algorithm can use any of PROB’s search methods (depth-first, breadth-first and the default mixed depth-first/breadth-first search). Especially for large state spaces, this increases the likelihood of detecting errors; see [Leu08].
- The algorithm of [LB05] computes the state space of the abstract model independently of the refined model and thus does not check the gluing invariant. Furthermore, in its current form, it cannot make use of Event-B’s witnesses and hence has to keep track of *sets* of states in the abstract model. Our algorithm makes use of the witnesses, which often allows us to only relate a single abstract state to a refined state. It also allows us to relate parameters of the refined event to parameters of the abstract event. E.g., in our running example, *fill_mug* has no parameter in *CoffeeR2*, while it has the parameter x in *CoffeeR1* and *CoffeeM*; the witness $x = level(clvl')$ in *CoffeeR2* tells us which parameter value to use in the abstract models. The algorithm of [LB05] was devised in a setting of CSP and classical B, where parameters of events or operations cannot be refined. E.g., in the running example it would report a counter-example (*INITIALISATION*, *fill_mug*) for *CoffeeR2*, unless we tell it to ignore all parameters in traces.

4.1.13 Other Tools for Animating Event-B

As already indicated above, the tools Brama and AnimB are also capable of performing multi-level animation of Event-B models, and have partially inspired this work. Unfortunately there is little scientific or technical documentation available for either of these tools. A few notable differences are

- Both Brama and AnimB require the user to specify explicitly values for constants; that is, we had to “calculate” the Cartesian products for the level constant in Fig. 4.2 by hand.
- PROB can be driven by a model checker to systematically search for errors, and to validate LTL formulas.

- PROB uses a constraint-solving approach to find solutions for predicates, while AnimB and Brama seem to rely on pure enumeration. As such, PROB can evaluate much more complicated guards and predicates than AnimB or Brama.

Another animator for Event-B is [ASAA08]; but it does not yet seem to support refinement animation. The same is true for the animator in [ABC⁺02] for classical B.

4.1.14 Conclusion

We have described a method for animation and validation of Event-B models. The key ingredient that makes the algorithm tractable are the witnesses of Event-B. We have implemented the algorithm within PROB, and have shown how a variety of refinement errors can now be detected effectively. We have applied the technique to various case studies, and have animated up to 14 levels simultaneously. The algorithm presented in the article also achieves further performance improvements over previous work based on trace refinement, for instance, by applying symmetry reduction to multiple levels of refinement at once.

Animation does not need to provide all capabilities that formal proof provides, because we consider proof and animation complementary techniques of validation. For instance, for finding candidates of invariants formal proof appears superior; for checking whether they are always satisfied, i.e., finding counter examples, animation appears superior. Moreover, animation can be used to reason about properties that have not (yet) been formalised.

To improve the capabilities of the animator, Event-B types and definitions would be needed that support animation better. For instance, transitive closures are defined as sets of all transitive closures of all relations of a certain type. For animation, it would be sufficient to compute only the closures of the relations actually appearing in a model. We would like to link the animator closer to the other components of the Rodin tool. We would also like to indicate certain errors found in Event-B models by PROB in associated proof obligations, for instance, so that the violated proof obligations can be marked as “not provable.”

machine *CoffeeM* sees *CofCtx*

variables *alvl*

invariants

@inv1 $alvl \in FILL$

variant $(\{full \mapsto 2, half \mapsto 1, empty \mapsto 0\})(alvl)$

events

event *INITIALISATION*

begin

@mf $alvl := empty$

end

event *fill_mug*

any *x* when

@g0 $alvl = empty$

@g1 $x \neq alvl$

then

@a1 $alvl := x$

end

convergent event *drink*

when

@g1 $alvl \neq empty$

then

@a1 $alvl : \in \{empty, half\} \setminus \{alvl\}$

end

end

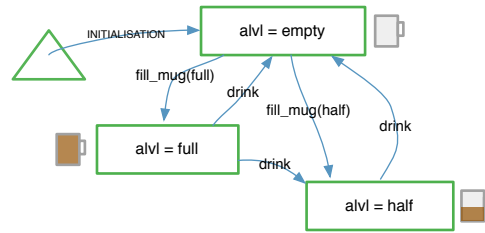


Figure 4.3: Coffee dispenser model and its state space

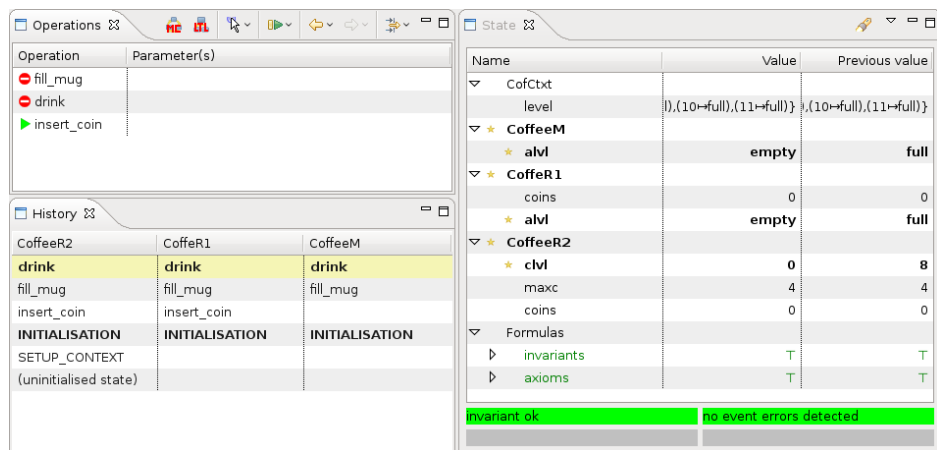


Figure 4.4: PROB operations and state view after the trace *insert_coin*, *fill_mug*, *drink*

```

context ImpCtxt

constants maxc

axioms @amc maxc  $\in \mathcal{N}_1$ 

end

```

machine *CoffeeR1* refines *CoffeeM* machine *CoffeeR2* refines *CoffeeR1*

sees *CofCtxt*

variables *alvl coins*

invariants

@ci *coins* $\in \mathcal{N}$

events

event *INITIALISATION*

extends *INITIALISATION*

begin

@ai *coins* := 0

end

event *fill_mug*

extends *fill_mug*

when

@gc *coins* > 0

then

@delc *coins* := *coins* - 1

end

convergent event *drink*

extends *drink*

end

anticipated event *insert_coin*

begin

@insc *coins* := *coins* + 1

end

end

sees *CofCtxt ImpCtxt*

variables *clvl coins*

invariants

@ifl *clvl* $\in 0 \dots 11$

@lvl *alvl* = *level*(*clvl*)

variant *maxc* - *coins*

events

event *INITIALISATION*

begin

@cci *coins* := 0

@fli *clvl* := 0

end

event *fill_mug* refines *fill_mug*

when

@gc2 *coins* > 0

@ml *clvl* $\in level^{-1}[\{empty\}]$

with

@x *x* = *level*(*clvl'*)

then

@delc2 *coins* := *coins* - 1

@ffl *clvl* $\in level^{-1}[\{full\}]$

end

convergent event *drink* refines *drink*

when

113 @dgfl *clvl* $\notin level^{-1}[\{empty\}]$

with

@alvl' *alvl'* = *level*(*clvl'*)

then

@dff *clvl* $\in level^{-1}[\{empty, half\} \setminus \{level(clvl)\}]$

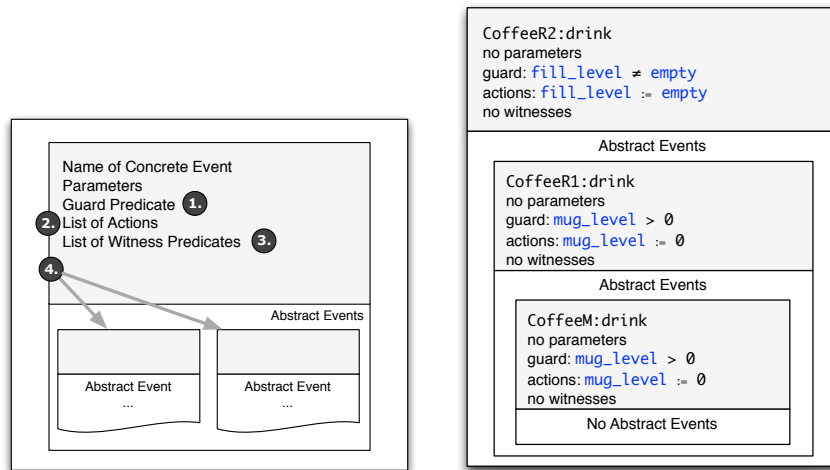


Figure 4.6: Illustration of the algorithm and one particular event structure

Machine	Step	values
<i>CoffeeR2</i>	1. evaluate the guards: @gc2: $coins > 0 \equiv 2 > 0 \equiv \top$ @ml: $clvl \in level^{-1}[\{empty\}] \equiv 0 \in 0..2 \equiv \top$	
<i>CoffeeR2</i>	2. action @delc2: $coins' = coins - 1 \equiv coins' = 2 - 1$ $coins' = 1$ action @ffl: $clvl' \in level^{-1}[\{full\}] \equiv clvl' \in 8..11$ $clvl' = 9$	
<i>CoffeeR2</i>	3. witness for the abstract parameter x : $x = level(clvl') = level(9) = full$	$x = full$
<i>CoffeeR2</i>	4. check the guard of the abstract event: @gc: $coins > 0 \equiv \top$	
<i>CoffeeR1</i>	2. action @delc: $coins' = coins - 1 \equiv 1 = 2 - 1 \equiv \top$	
<i>CoffeeR1</i>	3. (no witness to check)	
<i>CoffeeR1</i>	4. check the guards of the abstract event: @g0: $alvl = empty \equiv empty = empty \equiv \top$ @g1: $x \neq alvl \equiv full \neq empty \equiv \top$	
<i>CoffeeM</i>	2. action @a1: $alvl' = x$	$alvl' = full$
<i>CoffeeM</i>	3. (no witness to check)	
<i>CoffeeM</i>	4. no more abstract events, stop	

Figure 4.7: Example run of the refinement algorithm for event *fill_mug*

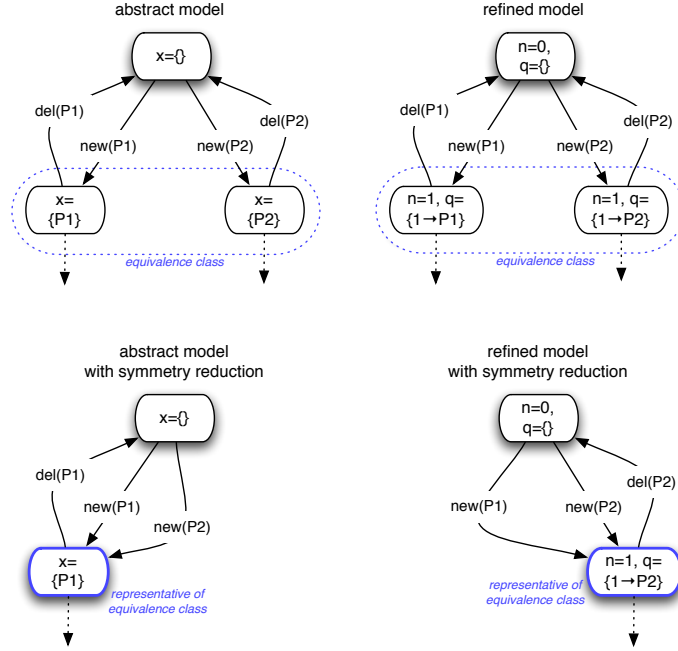


Figure 4.8: Illustration of symmetry reduction for trace refinement checking

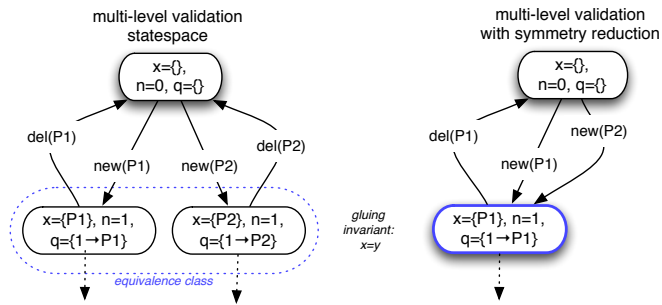


Figure 4.9: Illustration of symmetry reduction for multi-level validation

Chapter 5

Dependability

*D. Basin, J. Bryans, J. Fitzgerald, T. Hoang, A. Iliasov, C. Jones,
L. Laibinis, I. Lopatkin, M. Mazzara, Y. Prokhorova*

In the DEPLOY project, we sought to use formal methods to enhance the dependability of systems, in particular the types of systems developed by industrial partners. The pilot studies provided a base and a "reality check" to this work. In working with the industrial partners, an incompatibility we observed was in the languages and terms that we used. Classical dependability research is presented in terms of faults, errors and failures, but the language preferred by industrial partners (to describe the same phenomena) was terms such as exception or incompatibility. The subject however was clearly of interest to the industrial partners - an analysis of the resilience requirements of each of the pilot studies was carried out, which found that many of the requirements were related to fault-tolerance features.

In this chapter we provide general guidelines on modelling dependability requirements, on modelling exceptional/faulty behaviour and modelling fault tolerance mechanisms using flows. We also describe the use of a fault analysis technique (FMEA) together with representations of modes in Event-B. The chapter concludes with a brief overview of experience with modelling security properties in access control using Event-B.

5.1 Modelling dependability requirements

The objective of this section is reporting the main ideas which have been brought forward and developed on dependability requirements in the automotive domain by Newcastle and Bosch . A digested understanding of the

issues investigated over the duration of DEPLOY is presented in the attempt of offering a broad view of the research after the close and fruitful collaboration. The work reported here has been already (at least partly) published in a number of papers or deliverables and builds on top of other. In [Edi] the reader will find (in Chapter 7) a report of some of the problems which have been approached, although described in a seminal form and shaped by the understanding reached at that time. In [LGM⁺10] the first Bosch pilot, the Cruise Control, has shown power and limits of Problem Frames and Event-B, while [GGL⁺11] reports about the neat success of the second pilot, namely the Start Stop System, and the alternative approach taken. All this work is built on top of a solid theoretical investigation which originates from literature on Problem Frames ([Jac01] among the others) and the HJJ method [JHJ07]. The interested reader will find also useful the discussion on methods, the epistemology behind Problem Frames and HJJ and their root in modern logical positivism and the connection between all this and formal specifications [Maz09b], [Maz09a], [Maz11]. Finally, [BFJ⁺10] reports on how the industry problems have been tackled from the methodological point of view.

Collecting Requirements

Chapter 4 of D19 [LGM⁺10] is about Cruise Control Deployment. One of the achievements of this pilot was how the original informal requirements specification was managed. The specification was written in natural language (German) and it included ca. 250 different requirements, sometime duplicated and without a uniform use of the terminology. The most difficult challenge in the (re)organization of the requirements was the fact that some of them were implementation specific without a clear understanding and use of abstraction. As a consequence, the gradual introduction of Event-B style refinement was basically impossible. Section 4.3 of D19 describes in great detail the methodology applied to the original document to obtain a Semi-Formal specification representing an intermediate step toward an Event-B specification. The potential of Problem Frames was realized quite early in the project. Engineering the heterogeneous set of original requirements and emphasizing the distinction between the machine and the real world was where the Problem Frames approach showed its characteristic profile. Although a certain amount of arbitrariness cannot be completely eliminated in the specification phase, following the Problem Frames philosophy made the border between requirements and assumptions more clear.

Defining the Border

To define the border of what has been done and what has been not we have recall the Problem Frames distinction between normal design and radical design, as presented in [Vin93]. To briefly recall the concepts, normal design is the routinary traditional design, while radical stands for bringing innovation. To contextualize the Cruise Control pilot, we need to make two observations: first, the design and development of the Cruise Control was, by itself, something where the engineers have total confidence that the final outcome (product) will certainly work; second, the methodology developed for the (re)organization of the requirements was experimented on a pilot project (the Cruise Control) where the requirements were already (although poorly) collected and (someway) "organized". This prevented the experimentation on a case study where the requirements themselves had to be collected and structured in the first instance. As a consequence, we did not collect any evidence about the use of Problem Frames in radical design.

The Methodology

To cope with the complicated situation of a big number of preexisting and practically unstructured requirements, an innovative and elegant methodology were developed aiming at creating a hierarchy of Problem Diagrams by means of an alternate application of *Elaboration* and *Projection* that are novel reinterpretations of refinement and decomposition. Picture 4.2 and 4.3 of D19 [LGM⁺10] deserve attention for the reader who wants to appreciate the details of this process (in particular, Table 4.3 shows the mapping between Problem Frames and Event-B). It is worth noting how this semi-formal specification generated by the Elaboration/Projection process was not expressive enough to link Problem Frames to Event B. The key problems was its lack of proper behavioral information. Indeed, it is not clear how a general requirement should be mapped into one or more events in a repeatable (mechanizable) way without such a precious description or representation in a proper formalism (or formal language). Indeed, the requirements, although clearly structured, were still expressed in natural language and the overall system behavior was not described anywhere. This consideration showed the need for a formalism able to express the dynamic behavior, and here was where Finite State Machines were introduced and discussed. However, mapping the state machines structure to Event-B is still troublesome and this is what lead to a different approach in the second pilot, the Start Stop System [GGL⁺11].

Linking Problem Frames and Event-B

This was the most intriguing issue. On one side, the adoption of Problem Frames appeared intuitive and necessary to efficiently organize the requirements into something encodable into Event-B. On the other, even if both Problem Frames and Event-B offer perspective which are useful in the specification and development phases [LGM⁺10], they represent substantially different views of a system. Despite the intrinsic complication of finding a way to link the two, significant steps forward have been made to this respect and state machines ended up to be the key bridge between the two worlds during the development of both the pilots. Understanding how different philosophies are behind the two approaches was a key for the success. In Problem Frames, user requirements are seen as being about relationships in the operational context and not about functions the software system has to perform. Therefore, the focus is not immediately on the functional behaviour. It is a sharp change of perspective with respect to other requirements analysis techniques like the Use Case approach [Bit02] where the interface is specified and the focus is on the interaction user/machine. The key difference is the emphasis on describing the environment and not the machine or its interfaces. The overall philosophy is that the problem is located in the world and the solution in the machine. The entire Problem Frames software specification goal is modifying the world (the problem environment) through the creation of a dedicated machine which will be then put into operation in this world. The machine will then operate bringing the desired effects. Event-B has instead a different approach. Its purpose is to describe reactive systems with the help of events. Refinement is there used to move from one level of abstraction to the next one, using mathematical proof to verify consistency between the two levels. It is not difficult to see how the emphasis is more on the functional aspects of the machine. In [LGM⁺10] this issue is longly described and it is explained why there is no way to refine an Event-B model to get a Problem Diagram, or viceversa: it is not a matter of level of abstraction or refinement steps, the two represent two different projections of the same system. This is way it is necessary to find a middle layer to make the whole machinery work.

Bridging the Gap between Formal and Informal

The major focus of all DEPLOY (at least in WP1) has been about bridging the gap between the informal and formal world. The described extended version of Problem Frames has been used for the Cruise Control to brilliantly solve the problem of requirements engineering. The problem arose when

trying to map this structure based on elaboration and projection into an Event-B model. State machines were introduced to specify the Cruise Control behaviour, but the pilot was approached by going straight from Problem Frames to Event-B, without an intermediate specification, just mapping Problem Frames elements into Event-B elements [LGM⁺10] and then using refinement. Because of this, the model became complicated to understand since conditions for a single state machine transition had to be split and distributed over multiple events. Problems Frames do not provide any mean to describe state machines or behaviour in general. Extensions of Problem Frames have been considered [LGM⁺10], but finally another solution has been implemented. An external drawing tool had to be used and conditions had then to be synchronized with the requirements expressed in the Problem Frames model. As a consequence of the lesson learnt from this experience, the Start Stop System has been approached with the introduction of another phase having as input the Problem Frames model and the requirements document and as output a specification document describing the static structure and the dynamic behaviour of the system. The specification approach, described in detail in [GGL⁺11], is inspired mainly by RSML [LHHR94] which provide concepts based on state machines for the description of dynamic behaviour of software systems. Separating in this strict way requirements analysis and specification simplifies then the task of formal modelling.

5.2 Describing exceptional behaviour (i.e. fault modelling)

In this section bring together the work on describing the exceptional behaviour (or “faults”) carried out within DEPLOY. The analysis of the resilience requirements of each of the pilot studies is outlined. Work with SAP is presented, which considered a scenario where a choice had to be made between two sets of infrastructure guarantees in the implementation of a protocol design. We were able to demonstrate that one of the sets of guarantees was not strong enough to support the protocol as designed. We also discuss the work on describing modal systems.

5.2.1 Modelling middleware

The work summarised here is published in [BFRR09], in which we report our experience in using formal methods to enhance processes and tools for development of business information software based on service-oriented architectures. In our work, using the pilot studies provided by SAP, we focus

on the configuration of middleware, verifying application-level requirements in the presence of faults. We prove properties of models of business protocols and expose weaknesses of certain middleware configurations with respect to particular protocols. We then extended the approach to use models automatically generated from diagrammatic design tools, opening the possibility of seamless integration with current development environments. Increased automation in the verification process, through domain-specific models and theories, is a goal for future work.

The challenge we address is that of gaining the benefits of formal modelling and analysis in the development of SOA-based business information systems, while retaining a beneficial trade-off between the effort invested by developers insights that they gain. Our approach aims for a smooth transition between the traditional development processes and formal approaches. Initially, developers might not directly interact with a formal modelling tool, but continue to use pre-existing diagrammatic domain-specific modelling environments. The models developed in these environments could be automatically translated into a suitable formal notation and treated with automated analysis tools. While the insights gained from such purely automatic analysis might be less than those arising from a more thorough adoption, we expect that the formal methods would be seen by developers as a benefit demanding little additional effort. In the long run, we expect that subjectively experienced and objectively measurable benefits will lead to a positive attitude towards the methods and tools and then to their more extensive direct use.

The purpose of our study is to determine the technical feasibility of using Event-B/RODIN to support the analysis of design models of SOA-based business information applications. The specific focus is on selecting an appropriate configuration for middleware from among alternatives offering different levels of fault tolerance (EO or EOIO). The applications models should be derived automatically from existing graphical design tools and there should be a good level of automation in the analysis of the models.

Our approach is to use a series of case studies, with the aim of producing a proof-of-concept of the automated analysis discussed above. In our studies, we needed to:

1. Establish that Event-B/RODIN can indeed support the comparison of alternative middleware components with respect to application-level properties.
2. Define the process for interfacing alternative middleware models (e.g. EO or EOIO) to pre-existing application-level models.
3. Develop appropriate strategies for combining middleware models with

application models derived from the pre-existing graphical design tools so as to yield a good degree of automation in the analysis.

These are the subjects of the studies described in Sections 5.2.1 to 5.2.1 respectively. The studies used both the B2B and A2A pilot studies.

In the business-to-business (B2B) choreography (or protocol) [WRSC08] two components, a buyer and seller, exchange messages in order to negotiate the price of a product or service. The negotiation is initiated by a proposal from the buyer detailing purchase conditions such as price, quantity, or delivery date. The two parties may then arbitrarily exchange further proposals. A party indicates agreement to a proposal by returning that proposal. The negotiation may be cancelled at any time. The critical property that B2B is designed to establish is:

Property 1 *When a run of the protocol terminates, either the buyer and seller should have agreed to the same price, or they should agree that the negotiation has been cancelled.*

In the application-to-application (A2A) choreography two components interact to meet a requirement from a customer. The *ordering component* is responsible for managing customer requirements, and the *supply chain requirements component* coordinates the services used to process these requirements. The protocol starts when the supply chain component receives customer requirements from the ordering component. The supply chain component may then send notification of (partial) fulfilment of these requirements (e.g. delivery) back to the ordering component. The ordering component may also send queries and preliminary reservation requests and the supply chain component sends current supply planning and delivery information to the ordering component.

Study 1: Middleware Models

The aim of the initial study, using the B2B protocol, was to confirm that the Event-B/RODIN tools could support the comparison of alternative middleware components with respect to application-level properties. The application is built from a buyer and a seller component, and either EO or EOIO middleware. Our method was first to build Event-B models of EO and EOIO middleware, and an abstract model of the B2B protocol that did not contain an explicit component representing middleware. Each of the middleware models was composed in turn with the B2B model, and the Event-B/RODIN tools were used to compare the two combinations.

Each application-level event involves both a protocol party (buyer or seller) and the middleware. It is therefore partially described in each model.

The protocol model describes the effects of the action local to the buyer or seller and the middleware model describes the effects of the action local to the middleware. Composing the middleware with the protocol involves composing each of these actions.

For example, the application-level action of the buyer sending a proposal p to the seller appears in the protocol model as *Buyer_send* (see Figure 5.1).

```

Buyer_send
  any  p  where
     $p \in PROPOSAL$ 
     $p \notin \{empty, cancel\}$ 
     $p \neq last\_s\_o\_rec$ 
     $BAgreeStatus = NoAgreement$ 
     $BCancelStatus = NotCancelled$ 
  then
     $curr\_b\_o := p$ 
  end

```

Figure 5.1: Buyer_send in the protocol

Figure 5.2 shows the EO model invariants. These describe the structure of the middleware.

```

invariants:
  inv1   $mw\_to\_seller \in PROPOSAL$  functional  $\mathcal{N}_1$ 
  inv2   $mw\_to\_buyer \in PROPOSAL$  functional  $\mathcal{N}_1$ 

```

Figure 5.2: EO invariants

In the case where the proposal is not yet in the middleware, the local effects of the buyer sending a proposal are defined as *Buyer_send_mw*, shown in Figure 5.3. A separate event describes the case where the proposal being sent is already in the middleware.

Combining two events into one retains shared parameters (in this case p), and conjoins guards and actions. In this way, each event in the protocol


```

Buyer_send_mw
  any p where
     $p \notin \text{dom}(\text{mware\_to\_seller})$ 
  then
     $\text{mware\_to\_seller}(p) := 1$ 
  end

```

Figure 5.3: Buyer_send in EO middleware

model is combined with the appropriate event from the middleware model to create a model of the application. This process is automated by a composition plugin available for the Event-B tool [Sil].

When using the EO middleware, the critical property is formulated as is Figure 5.4. When the B2B protocol runs on EO middleware, this critical

invariants:

```

inv20( $\text{BuyerAgStatus} = \text{Agreement} \wedge$ 
 $\text{SellerAgStatus} = \text{Agreement} \wedge$ 
 $\text{mware\_to\_buyer} = \emptyset \wedge$ 
 $\text{mware\_to\_seller} = \emptyset$ )
 $\Rightarrow \text{curr\_b\_o} = \text{curr\_s\_o}$ 

```

Figure 5.4: Property 1 in EO

property cannot be proved.

Study 2

In this study our aim was to further investigate how to develop models of business applications which allow for the introduction of middleware representations from a range of components and to develop standards for the integration of middleware into application models.

In this study, we integrated the middleware models with an independently developed model of the B2B protocol. This allowed us to identify more clearly

the interface between the middleware and the protocol parties, and to develop a set of guidelines for protocols developers wishing to use the middleware.

The identified guidelines include:

- **protocol parties** should be developed in one machine, with no representation of middleware. Each send or receive event should instead use a reserved variable name as a parameter (e.g. “*p*” in Section 5.2.1).
- **correctness criteria** for the protocol should be expressed as application invariants, (although they will not in general be provable before a middleware model is integrated).
- **complex message sets** should be defined in a new context visible to both the protocol model and the middleware model.

Study 3

In the final study, we moved closer to our goal of using Event-B models that had been automatically generated from existing graphical modelling tools used in SAP, rather than hand-crafted. As might be expected, a major challenge here is in achieving a good degree of proof automation with auto-generated models, so it is necessary to consider alternative ways of introducing the middleware into the refinement chain in order to gain insights into their benefits and disadvantages.

We used two different techniques to produce a new machine containing EOIO middleware, illustrated in Figure 5.5. The first technique produced the new machine by refining a machine in the original model containing the abstract choreography, and the second by refining a machine containing the low-level behaviour of the protocol.

An Event-B model of the A2A protocol was automatically generated from existing diagrammatic domain-specific modelling languages, rather than hand-crafted. It contained two machines, `m_choreography` and `EO_A2A`. The first machine, `m_choreography`, had a high-level view of behaviour and no explicit component representing the middleware. It contained seven events and two invariants, and produced no proof obligations. The second machine, `EO_A2A`, contained the local behaviour of the ordering and supplier components and a model of EO middleware. In `EO_A2A` each of the events from `m_choreography` was refined by a “send” and a “receive” event, giving 14 events. It had 22 invariants and 268 proof obligations of which 263 were proved automatically and 5 required (trivial) intervention.

To investigate the modelling options, two machines containing EOIO middleware were developed by hand. `EOIO_A2A_ONE` was a refinement of

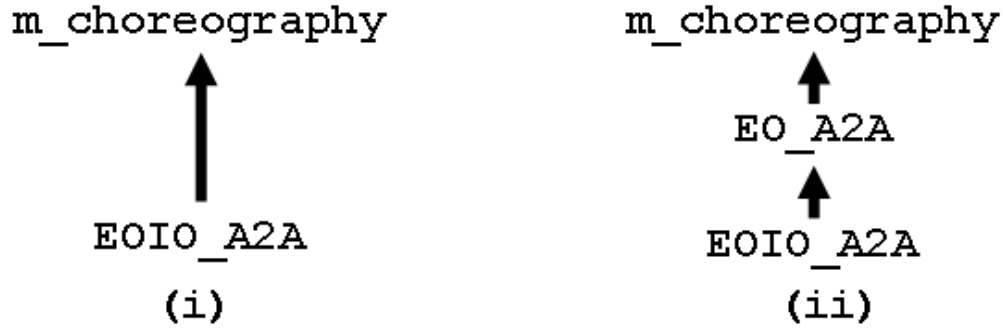


Figure 5.5: Two refinement techniques

`m_choreography` and `EOIO_A2A_TWO` was a refinement of `EO_A2A`.

There were 257 proof obligations in `EOIO_A2A_ONE`, 162 of which were proved automatically. The remaining 95 were significantly more complex than the invariants in `EO_A2A`. The primary source of the increased complexity was ten invariants that relate messages in middleware to states within the machine. In `EO_A2A` the process components exchange messages from the set *MESSAGES*. The EO middleware does not offer an ordering guarantee, so the representation is an unordered bag *channel* of type $MESSAGES \rightarrow \mathcal{N}$. The quantity of a message *M* in the middleware is given as *channel*(*M*).

`EOIO_A2A_TWO` contains 186 proof obligations, of which 116 were proved automatically. The remainder required about 6 hours of effort. The bulk of this effort went in proving the linking invariants between the two middleware representations.

The benefit of the first approach is that there is no need for the local machine `EO_A2A`. It represents the case where a machine containing a representation of EO middleware is not available, or the developer is interested exclusively in EOIO middleware. The (overwhelming) disadvantage is that the level of manual intervention required to prove the proof obligations is too high. Conversely, the second approach requires an intermediate machine (`EO_A2A`) to be built and proved. It represents the case where a machine containing a representation of EO middleware is available to be used.

5.2.2 Resilience Requirements in Pilot and Minipilot Studies

We have analysed the informal requirements descriptions produced by deployment partners for the pilot and mini-pilot studies in Deploy¹ with the focus on fault tolerance features (mainly error recovery and fault assumptions). These aspects represent a substantial proportion of the requirements (sometimes as much as 35-40%). The major source of faults considered in these systems is the environment, including sensors, external networks and human operators. Dealing with software design faults is never stated as a requirement, and only rarely do requirements define hardware faults (e.g. node crashes in a distributed application) and means to handle them.

The system requirements include descriptions of degraded functionality, the most typical example being system safestop. More generally we observe that the requirements always include information about how general system behaviour is affected by various abnormal situations. Unfortunately this information is rarely explicitly stated as a priority requirement (sometimes, we needed to deduce this information from other requirements).

We have found that nearly all system requirements state the operational conditions resulting in different intended functionalities provided by the system. The conditions and functionality together comprise the concept of *operational mode* [DIRR09, IRD09]. Descriptions of system modes and mode transitions are often intertwined with error recovery; sometimes this includes fault handling by system degradation.

A final observation is that requirements related to error recovery are not structured in a way that makes it easy for modellers to work with these issues. The relevant requirements are typically scattered over the whole requirements document and refer to issues related to different levels of abstractions. For example, none of the documents reviewed had a table of fault assumptions.

We conducted a work on fault tolerance views of the system [LIR10] to facilitate development of FT features and improve traceability of resilience requirements.

5.3 Fault-tolerance in Design

This Section presents two modelling environments developed for Event-B, and shows their relevance to fault-tolerant design. It begins with a discus-

¹The detailed requirements documents for the case studies are mostly confidential, but descriptions of the pilots are provided in public deliverables for the deployment workpackages.

sion of the *Flow* modelling environment, and contains a description of a tool designed to model the flow of control through an Event-B model. The second part (on fault tolerance views) is again a description of a modelling environment, this time designed to aid in the construction of fault tolerance features in a concise manner and to formally link them to Event-B models.

5.3.1 Flows

Depicting the evolution of a system is the focus of any state-based formal design. In Event-B this is achieved with events - elementary state transformation. An Event-B event is the smallest and the only unit of a change in a model. Such fine-grained approach to specifying state transitions results in a simple and efficient proof semantics and delivers a higher rate of verification automation compared to a close relative, Classical B. A streamlined proof semantics comes at a price as becomes evident in a large Event-B development. While the verification part generally scales well (most proof obligation kinds require instantiations in a linear proportion to the number of events) large models become difficult to read, navigate about and communicate. Although these drawbacks may be seen as 'soft' properties of a model they must be taken seriously; afterall, an unreadable model has next to no value as a design document.

The Flow plug in strives to alleviate some of these problems by offering a high-level visual notation to reason about event orderings. The flow approach comprises two main methodological strands: explicit specification of control flow and the proof of related properties for systems describing a succession of computation steps (as opposed to purely event-based, data-driven designs); formulation of use case scenarios that act as a cross-check for the functionality implemented in a model.

To offer a guarantee that the prescribed event ordering is present in the model the plug in generates proof obligations - additional theorems that must be discharged alongside with traditional consistency and refinement conditions. The proof-based approach fits well the Event-B development method and there is an ongoing work to enable flow-driven model animation.

Explicit formulation of event orderings frees one from having to define auxiliary variables to establish that the presumed paths are present in the model. Moreover, the kind of proof obligations the Flow plug in generates would be most tedious and error-prone to construct by hand but they are far more economical in the terms of proof effort than a more naive approach with auxiliary variables acting as program counters. Further, it is possible to construct theorems about states that a system reaches by taking a certain path. Such theorems do not easily translate into the core Event-B notation.

The second direction, use case scenarios, is a popular technique for the validation of software systems and constitute an important part of requirements engineering process. It is an essential part of the description of functional requirements of a system. It is assumed that a library of use case scenarios is available together with a system requirements document and use cases are presented in a sufficiently precise manner. A modeller expresses use case scenarios as formal verification conditions that become a part of a formal model of a developed system. It is guaranteed that the final product obtained from such a model possesses, by the virtue of the development method, all the properties expressed in use cases scenarios. The overall approach is generally in line with some existing work on formalisation and formal validation of use cases. The approach is investigated on the basis of an extension of the Event-B modelling method with a technique for formally capturing use case scenarios as theorems over model state.

Flow proof semantics This section gives a summary of the main proof obligations behind the flow approach. Consider the following relations over pairs of relations on some set S .

$$\begin{aligned} U &= \{f \mapsto g \mid \emptyset \subset f \subseteq S \times S \wedge \emptyset \subset g \subseteq S \times S\} \\ \mathbf{ena} &= \{f \mapsto g \mid f \mapsto g \in U \wedge \text{ran}(f) \subseteq \text{dom}(g)\} \\ \mathbf{dis} &= \{f \mapsto g \mid f \mapsto g \in U \wedge \text{ran}(f) \cap \text{dom}(g) = \emptyset\} \\ \mathbf{fis} &= \{f \mapsto g \mid f \mapsto g \in U \wedge \text{ran}(f) \cap \text{dom}(g) \neq \emptyset\} \end{aligned}$$

The definitions are concerned with the properties of a composite relation $g \circ f$. $f \mathbf{ena} g$ states that $g \circ f$ is defined for every value on which f is defined - $\text{dom}(g \circ f) = \text{dom}(f)$; relation $f \mathbf{dis} g$ implies that $g \circ f = \emptyset$; $f \mathbf{fis} g$ means that there is at least one pair of values satisfying relation $g \circ f$: $\exists v, u \cdot u (g \circ f) v$. The relations enjoy the following properties.

$$\begin{aligned} \mathbf{dis} \cap \mathbf{fis} &= \emptyset \quad \Leftrightarrow \quad \neg(f \mathbf{dis} g \wedge f \mathbf{fis} g) \\ \mathbf{ena} \cap \mathbf{dis} &= \emptyset \quad \Leftrightarrow \quad \neg(f \mathbf{ena} g \wedge f \mathbf{dis} g) \\ \mathbf{dis} \cup \mathbf{fis} &= U \quad \Leftrightarrow \quad f \mathbf{dis} g \vee f \mathbf{fis} g \end{aligned}$$

Let p_e, G_e, R_e characterise the parameters, guard and action of an event e . Assuming that the consistency proof obligations are discharged, the universe of system states Σ is said to be its safe states: $\Sigma = \{v \mid I(v)\}$. An event e is a next-state relation of the form $e \subseteq \Sigma \times \Sigma$. Let $\mathbf{before}(e) \subseteq \Sigma$ and $\mathbf{after}(e) \subseteq \Sigma$ signify the domain and the range of relation e . The set $\mathbf{before}(e)$ corresponds to the enabling states defined by the event guard and $\mathbf{after}(e)$ is a set of possible new states computed by the event:

$$\begin{aligned}\text{before}(e) &= \{v \mid I(v) \wedge \exists p_e \cdot G_e(p_e, v)\} \\ \text{after}(e) &= \{v' \mid I(v) \wedge \exists p_e \cdot (G_e(p_e, v) \wedge R_e(p_e, v, v'))\}\end{aligned}$$

Let b and h be some events. Taking into the account the definitions of **before** and **after**, relations **ena**, **dis**, **fis** may be expanded as follows.

$$\begin{aligned}b \text{ ena } h &\Leftrightarrow \text{after}(b) \subseteq \text{before}(h) \\ &\Leftrightarrow \{v' \mid I(v) \wedge \exists p_b \cdot (G_b(p_b, v) \wedge R_b(p_b, v, v'))\} \\ &\subseteq \{v \mid I(v) \wedge \exists p_h \cdot G_h(p_h, v)\} \\ &\Leftrightarrow \forall v, v', p_b \cdot I(v) \wedge G_b(p_b, v) \wedge R_b(p_b, v, v') \Rightarrow \exists p_h \cdot G_h(p_h, v') \quad (\mathbf{FENA})\end{aligned}$$

$$\begin{aligned}b \text{ dis } h &\Leftrightarrow \text{after}(b) \cap \text{before}(h) = \emptyset \\ &\Leftrightarrow \text{after}(b) \subseteq \Sigma \setminus \text{before}(h) \\ &\Leftrightarrow \text{after}(b) \subseteq \{v \mid I(v)\} \setminus \{v \mid I(v) \wedge \exists p_h \cdot G_h(p_h, v)\} \\ &\Leftrightarrow \text{after}(b) \subseteq \{v \mid I(v) \wedge \forall p_h \cdot \neg G_h(p_h, v)\} \\ &\Leftrightarrow \forall v, v', p_b, p_h \cdot I(v) \wedge G_b(p_b, v) \wedge R_b(p_b, v, v') \Rightarrow \neg G_h(p_h, v') \quad (\mathbf{FDIS})\end{aligned}$$

$$\begin{aligned}b \text{ fis } h &\Leftrightarrow \text{after}(b) \cap \text{before}(h) \neq \emptyset \\ &\Leftrightarrow \exists v, v', p_b, p_h \cdot I(v) \wedge G_b(p_b, v) \wedge R_b(p_b, v, v') \wedge G_h(p_h, v') \quad (\mathbf{FFIS})\end{aligned}$$

Conditions (**FENA**, **FDIS**, **FFIS**) are the main flow verification conditions. Event-B proof obligations of these form are automatically derived by a tool supporting the approach.

Assumptions and assertions It is convenient to construct new events from existing events. We define two operators for this: *assumption* and *assertion*. An assumption constraints the enabling set of an event while an assertion constraints the set of new states computed by the event.

$$\begin{aligned}P.e &:= \{t \mapsto r \mid t \mapsto r \in e \wedge t \in P\} & \text{before}(P.e) &= \text{before}(e) \cap P \\ e.Q &:= \{t \mapsto r \mid t \mapsto r \in e \wedge r \in Q\} & \text{after}(e.Q) &= \text{after}(e) \cap Q\end{aligned}$$

It is trivial to see that such constrained events are safe: $P.e \subseteq e \wedge e.Q \subseteq e$. One special case is a constraint used in the roles of both assumption and assertion: $e(D) = D.e \cap D.e$. It is easy to see that $e(D)$ may be obtained by adding a guard predicate L to event e such that $D = \{v \mid L(v)\}$. It is often necessary to make statement about a state rather than an event. This

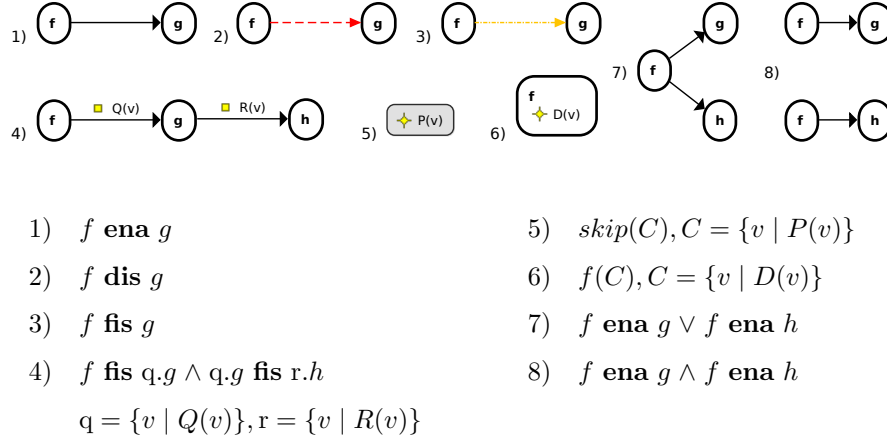


Figure 5.6: A summary of the core flow notation and its interpretation.

is done with the help of the SKIP event that does not change system state: $\text{SKIP} = \text{id}(\Sigma)$. Then $\text{SKIP}(D)$ is a stuttering step constrained to states D .

Graphical notation The approach is realised by a tool employing a visual, diagrammatic depiction of Flow theorems. A Flow diagram always exists in an association with one Event-B model. The theorems expressed in a Flow diagram are statements about the behaviour of the associated model. The basic element of a diagram is an event, visually depicted as a node (in Figure 5.6, f and g represent events). Event definition (its parameters, guard and action) is imported from the associated Event-B model. One special case of node is SKIP event, denoted by a grey node colour (Figure 5.6, 5). Event relations **ena**, **dis**, **fis** are represented by edges connecting nodes ((Figure 5.6, 1-3)). Depending on how a diagram is drawn, edges (flow theorems) are said to be in *and* or *or* relation (Figure 5.6, 7-8). New events are derived from model events by strengthening their guards (a case of symmetric assumption and assertion) (Figure 5.6, 6). Edges may be annotated with constraining predicates inducing assertion and assumption derived events (Figure 5.6, 4). Not shown on Figure 5.6 are nodes for the initialisation event start (circle) and implicit deadlock event stop (filled circle). The diagrams like those in Figure 5.6 (except 5 and 6 which are next-state relations rather than relations over events) are translated into theorems and appear as additional *proof obligations* for the associated Event-B model. A change in the diagram or Event-B model would automatically lead to the recomputation of affected proof obligations. Flow proof obligations are dealt with, like all other proof obligation types, by a combination of automated provers and interactive proof. Like in proofs of model consistency and refinement, the feedback

from an undischarged Flow proof obligation may often be interpreted as a suggestion of a diagram change such as an additional assumptions or assertion.

5.3.2 Fault Tolerance Views

The Fault Tolerance (FT) Views is a modelling environment for constructing fault tolerance features in a concise manner and formally linking them to Event-B models. It provides fault tolerance modelling facilities explicitly supporting the traceability of the FT and dependability requirements. The FT View is a special case of the Mode View, developed in our previous work on modelling modal systems [DIRR09]. It is essentially an application of the Mode View approach to fault tolerance. The FT View approach extends the Mode View with additional fault tolerance semantics, structural checks, and helps the modeller by offering reusable refinement templates. The details of the FT View approach can be found in [LIR10].

A mode/FT view is a graph diagram developed alongside an Event-B model which contains modes and transitions along with additional information necessary for establishing a formal connection with the model. The two basic concepts of the Mode View are *mode* and *transition*. Mode is a general characterisation of a system behaviour. It describes the functionality of a system and the operating conditions under which the system provides this functionality. A system switches from one mode to another through a mode *transition*.

The FT View adds two types of transition specialisation: an *error* and a *recovery* transitions. Relative to the transition and its type, we differentiate the FT types of modes: we say that an error originates in a *normal mode* and leads to switching to a *degraded mode* or a *recovery mode*. The recovery transition leads from the recovery mode back to normal. The distinction between the degraded and recovery modes is that the recovery mode is obliged to terminate and pass control back to the mode from which the initiating error originated. Safe-stop is regarded as a special case of a degraded mode.

Diagrams are built in a step-wise manner, starting from the most primitive case and introducing details using our *detailisation* process. The FT Views development process is a chain of documents similar to Event-B models. FT diagrams are built by incrementally adding new modes, errors and recoveries using the provided templates, and proving the refinement relationship between each two consequent views. Description of the *detailisation* process and templates can be found in [LIR10].

For the FT diagrams approach to be truly useful, there is a formal relationship between a diagram and an Event-B model establishing that a model

agrees with a diagram. The formalisation approach is based on a more general notion of formal modal systems [IRD09]. There is a study on linking mode diagrams and Event-B [DIRR09] and some of the results are reused for the FT Views.

We use the terms *assumption* to denote the different operating conditions and *guarantee* to denote the functionality ensured by the system under the corresponding assumption. Formally, a mode is characterised by a pair A/G where:

- $A(v)$ is an assumption - a predicate over the current system state;
- $G(v, v')$ is a guarantee, a relation over the current and next states of the system; and
- vector v is the set of model variables.

With assumption and guarantee of a mode being predicates expressed on the variables of an Event-B machine, we are able to impose restrictions on the way modes and transitions are mapped into model events and thus cross-check design decisions in either part.

A system switches from one mode into another through a mode transition that non-deterministically updates the state of v in such a way that the assumption of the source mode becomes false while the assumption of the target mode becomes true. Let us consider two modes, i and j . A mode transition is required to establish a new state v' such that $\neg A_i(v')$ and $A_j(v')$ while it is not under the obligation to respect $G_i(v, v')$. Mode transitions are also mapped into groups of events each of which must implement an instantaneous transition action.

With the basic formal framework of modes in place, it is possible to define consistency conditions for an FT view diagram and Event-B machine. The core principle is seeing the diagram as a source of further proof obligations for a machine. Thus, a simpler diagram leads the development of a machine through additional constraints.

A diagram is linked with an Event-B model by attributing a list of Event-B model events to each mode:

$$\begin{aligned}
 A_1/G_1 &\mapsto E_1 \\
 A_2/G_2 &\mapsto E_2 \\
 &\dots \\
 A_n/G_n &\mapsto E_n
 \end{aligned}$$

Event sets $E_1, \dots, E_n$ may overlap but should not be identical. The latter is due to the fact that two modes $A_i/G_i \mapsto E$ and $A_j/G_j \mapsto E$ are equivalent to a single mode $A_i \vee A_i/G_i \wedge G_j \mapsto E$ and thus there is no advantage in allowing configurations where modes have identical event sets.

The result of mapping modes into event sets is that there are now additional requirements to machine events: an event related to some mode must respect the mode guarantee provided the mode assumption holds. Execution cannot progress if there is no suitable enabled event for a mode. From the above the following conditions are derived.

All the events of a mode must satisfy the mode guarantee provided the assumption holds:

$$I(v) \wedge A(v) \wedge H(v) \wedge R(v, v') \rightarrow G(v, v') \quad (5.1)$$

The partitioning of the events into modes must be in agreement with the event guards. When an event is enabled then the assumption of its mode must hold. Since an event is potentially associated with multiple modes, the disjunction of all the relevant assumptions must hold:

$$H(v) \rightarrow A_1(v) \vee \dots \vee A_k(v) A_{k+1}(v) \vee \dots \vee A_n(v) \rightarrow \neg H(v) \quad (5.2)$$

where $A_1, \dots, A_k$ are the assumptions of the modes containing an event with guard $H(v)$ and $A_{k+1}, \dots, A_n$ are those not containing the event. The full list of the proof obligations is provided on a Mode/FT Views wiki page [moda].

We believe that the proposed approach represents a useful technique for disciplined and step-wise development of dependable systems and ensures an expressive link with the underlying Event-B formalism.

5.4 Derivation and Formal Verification of a Mode Logic for Layered Control Systems

5.4.1 Introduction

To cope with complexity, control systems are often developed in a layered fashion, which provides the designers with a convenient mechanism for structuring the system behaviour according to the identified architectural layers. The dynamic behaviour of a complex system is frequently defined in terms of operational modes – mutually exclusive sets of the system behaviour. The

system modes together with transitions between them are called a *mode logic* [LPS⁺97].

While designing layered mode-rich systems, we should ensure mode consistency and guarantee that the mode logic also caters to fault tolerance. In this section we propose to conduct Failure Modes and Effects Analysis (FMEA) of each operational mode to identify mode transitions required to implement fault tolerance. Fault tolerance is achieved in two different ways – by transitions to more degraded modes and system reconfiguration using redundant components. In this section we investigate a complex interplay between the state of components during reconfiguration and system modes. We propose a formal approach to designing complex fault tolerant mode-rich systems by refinement in Event-B. We demonstrate how to derive the mode logic of such systems and formalise it at different architectural layers.

We use the Event-B framework [Abr10] as our modelling language and the Rodin Platform [Rodb] as an automated tool support. The general idea of the development process presented in this section is to start from an abstract specification of the upper architectural layer and gradually unfold lower layers by refinement. Since refinement allows us to develop a system in a correct-by-construction fashion, stepwise unfolding of the architectural layers also guarantees preserving mode consistency between lower and upper layers.

This section is structured as follows. In Section 5.4.2 we present general guidelines for deriving a mode logic of complex layered systems. Section 5.4.3 describes our formalisation of fault-tolerant mode-rich systems and their properties. In Section 5.4.4 we illustrate our approach by an example – Attitude and Orbit Control System (AOCS). Finally, in Section 5.4.5 we discuss related work and give concluding remarks.

5.4.2 Deriving a Mode Logic

Layered Mode-Rich Systems

Often a layered architecture is used to facilitate development of complex control systems [Rub95]. It allows the designers to structure the system behaviour according to the identified abstraction levels. The lowest layer usually consists of the components (often called *units*) that work directly with hardware devices. The layer above contains the components encapsulating the lowest layer units by providing abstract interfaces to them. Depending on the system complexity and design decisions, there might be several intermediate layers. Finally, the top component provides an interface to the overall system.

Operational modes structure the dynamic behaviour of components at

different layers of abstraction. Leveson et al. [LPS⁺97] define an (*operational*) *mode* as a mutually exclusive set of system behaviours. A *mode logic* includes all the available modes and the rules for transitioning between them [LPS⁺97].

Each software component in layered mode-rich systems can be viewed as a mode manager. For simplicity, let us consider a two-layer system. It consists of the lower layer mode managers called *Unit Managers (UMs)*, which are monitored and controlled by the top layer *Mode Manager (MM)*.

We identify system modes according to the system operational constraints. We assume that the system executes a certain *scenario* defined in terms of its global modes. The scenario usually describes a sequence of modes leading to the most advanced state of the system (e.g., the state where it delivers the richest set of services). However, occurrence of failures may prevent the system from straightforward implementation of the scenario. This requires an introduction of the fault tolerance mechanisms into the system design. They are implemented as backward mode transitions (rollbacks) within the predefined scenario.

The dynamic behaviour of the overall system is cyclic. At each cycle MM monitors the current state of the lower layer units. If units are fault free then MM either completes the ongoing mode transition or maintains the current mode. It might also initiate the forward transition according to the scenario. If some failure that cannot be locally handled by UMs occurs, MM starts a backward mode transition.

An execution of the mode scenario by MM corresponds to certain mode transitions initiated by UMs. Mode consistency conditions are defined as interdependencies between global and local modes. Specifically, each global mode of MM may correspond to one or several combinations of the local modes of UMs. While the nominal part of a mode logic is usually well understood and given, the designers need to derive the fault tolerance part of the mode logic, i.e., the set of backward transitions to execute error recovery.

In the next section we demonstrate how to define the fault tolerance part of the mode logic in a well-structured way using FMEA as well as to introduce fault tolerance mechanisms into the mode logic.

Deriving a Mode Logic Using FMEA

Let us assume that the scenario (shown in Fig. 5.7 by solid arrows) defines how to bring a system from the non-operational mode GM_0 to the most advanced mode GM_n . Such a forward mode transition scenario is usually given in the system requirements document.

Moreover, let us assume that the implementation of the mode transition

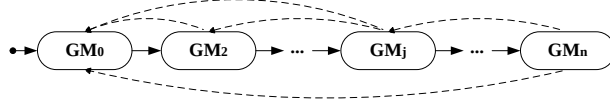


Figure 5.7: Mode transition scenario.

scenario can be interrupted either by transitional errors (i.e., errors that appear during a mode transition step) or unit usability errors (i.e., errors that occur when a unit performs below its required level).

In this section we consider units, i.e., the lowest layer components, to be dynamically reconfigurable. They consist of a main device (the nominal branch) and a spare device (the redundant branch). A unit encapsulates the errors and the reconfiguration state of its branches. In particular, if an error occurs in the nominal branch of the unit, it handles this error by switching to the redundant branch. The units have a special attribute called *status*. The unit status is *Locked*, when it is in an operational state and there is no ongoing reconfiguration, and *Unlocked* otherwise.

When MM chooses a new target mode, it initiates the corresponding mode transitions in the lower layer UMs. If an error is detected, the corresponding unit manager assesses the error and either initiates error recovery by itself or propagates the error to MM. MM, in its turn, makes a decision how to handle such an error. This decision usually involves rolling back to some less advanced (i.e., degraded) mode, as shown by dashed arrows in Fig. 1.

To systematically define the rollback procedures for each mode, we propose to conduct Failure Modes and Effects Analysis (FMEA) [FMEb, Lev95, Sto96]. FMEA is a well-known inductive safety analysis technique. For each system function or component, it defines possible failure modes, local and system effects, as well as detection and recovery procedures. The information is collected in a table form.

The traditional FMEA allows us to discover and structure failure modes of components. In this section we propose to conduct FMEA of each operational mode. We tailor FMEA to fit our purposes. In particular, we introduce an additional structure into the description of remedial actions. Namely, the remedial action field now contains three new subfields "Target mode", "Preconditions", and "Actions", describing respectively a new target mode, the conditions when the rollback to this mode should be initiated, and the system remedial actions in terms of new local mode transitions and, if needed, reconfiguration actions. If there are several preconditions given for the same target mode, the respective conditions should describe mutually exclusive situations.

For instance, Fig. 5.8 shows two excerpts from a FMEA worksheet of the

operational mode GM_j . The excerpts cover the situations when a hardware failure occurred in the unit U_i with and without an available spare respectively. Here, GM_{j-l} is some more degraded global mode, and GM_0 is the initial global mode.

Mode	GM_i		
Failure mode	Unit U_i failure with an available spare		
Possible cause	Hardware failure		
Local effects	Reconfiguration between unit branches. Change of unit status		
System effects	Remain the current global mode		
Detection	Comparison of received data with the predicted one		
Remedial action	Target mode	Precondition	Action
	GM _j	A state transition error in the nominal branch of U_i .	For a nominal branch of unit U_i , the status is set to Unlocked, and reconfiguration between branches is initiated.
		Insufficient usability of a selected nominal branch of U_i .	

Mode	GM_i		
Failure mode	Unit U_i failure without an available spare		
Possible cause	Hardware failure		
Local effects	Loss of preciseness in unit output data. Change of unit status		
System effects	Switch to a degraded mode		
Detection	Comparison of received data with the predicted one		
Remedial action	Target mode	Precondition	Action
	GM _{j-1}	A state transition error in the redundant branch of U_i . No state transition error in the redundant branch of U_k .	For unit U_i , any ongoing unit reconfiguration is aborted. For each branch in unit U_i , the status is set to Unlocked, and a state transition to non-operational state is initiated.
		Insufficient usability of a selected redundant branch of U_i . No branch state transition error. No problem on the redundant branch of U_k .	
	GM ₀	A state transition error in the redundant branch of U_k .	For each unit, any ongoing unit reconfiguration is aborted. For each branch in each unit the status is set to Unlocked, and a state transition to non-operational state is initiated.
		Insufficient usability of a selected redundant branch of U_k . No branch state transition error.	

Figure 5.8: FMEA worksheets.

FMEA of each system mode allows us to obtain a systematic textual description of all the procedures required to detect unit errors and perform their recovery. The main procedure of rolling back is as follows.

If a single unit fails then MM sets the new target mode to a degraded, however as advanced as possible, mode where the failed unit is not used, i.e., it is in a non-operational (*Off*) local mode. If several units fail then MM puts the system into a global mode where all failed units are in the *Off* modes.

In this section we have demonstrated how to systematically derive the fault tolerance part of the mode logic. However, it is easy to note that, even for a simple system, backward mode transitions significantly complicate the mode logic. Therefore, there is a clear need for techniques that facilitate not only derivation of the mode logic but also verification of its correctness. In the next section we demonstrate how to formalise the mode logic and verify it in Event-B.

5.4.3 Formalising a Mode Logic

Our formalisation aims at defining interdependencies between global modes, unit modes and their status as well as verifying mode consistency between different system layers.

We introduce a set of all possible modes, *MODES*. A relation on *MODES*, called *Scenario*, formally defines forward mode transitions, i.e., transitions starting from a non-operational mode and leading towards a fully operational one:

$$Scenario \in MODES \leftrightarrow MODES \quad (5.3)$$

All the possible forward transitions are represented by the relation *Mode_Order*:

$$Mode\_Order = Scenario^*$$

where $*$ is the transitive closure operation. Essentially, *Mode_Order* is a partial order relation on modes.

While *Scenario* formally represents the nominal part of a mode logic, the inverse relation *Mode_Order*[~] defines all possible rollbacks.

The mode to which the system should rollback to execute error recovery can be defined as a function *Error_Handling*:

$$\begin{aligned} Error\_Handling : CurrMode \times \\ (UnitErrors_1 \times \dots \times UnitErrors_n) \rightarrow MODES \end{aligned} \quad (5.4)$$

where *CurrMode* is the current global mode of the system and *UnitErrors*₁ ... *UnitErrors*_n are all the detected errors of units. For a mode transition to be completed, certain mode entry conditions should be satisfied. To formally define this, we introduce a function *Mode_entry_cond*:

$$\begin{aligned} Mode\_entry\_cond : MODES \rightarrow \\ \mathcal{P}(UnitModes_{1i} \times \dots \times UnitModes_{mj}) \end{aligned}$$

where *UnitModes*_{1i} ... *UnitModes*_{mj} are the local modes *i* ... *j* of the monitored units 1 ... *m*. For each global mode, the function returns a set of all valid combinations of unit modes.

The global mode transition cannot be started or completed if there is an ongoing reconfiguration of any unit in the system. To model this, we define an attribute *In_Reconfig*:

$$In\_Reconfig = Reconfig(U_1) \vee \dots \vee Reconfig(U_m)$$

where $Reconfig(U_1) \dots Reconfig(U_m)$ are the corresponding reconfiguration flags of the units $1 \dots m$. If there is no ongoing reconfiguration in the system, the flag $In\_Reconfig$ is *FALSE*, otherwise it is *TRUE*.

Reconfiguration is enabled only if an error of the nominal branch of a unit is detected. To reflect this, for each unit i , we introduce a function $Unit\_reconf_i$:

$$Unit\_reconf_i : Unit_i\_ERRORS \times BRANCH \rightarrow BRANCH$$

defined by its properties

$$\begin{aligned} \forall x \cdot Unit\_reconf_i(x \mapsto Branch\_A) &= Branch\_B \\ \forall x \cdot Unit\_reconf_i(x \mapsto Branch\_B) &= Branch\_B \end{aligned}$$

where $Unit_i\_ERRORS$ represents the errors of a particular unit i . Here $Branch\_A$ is the nominal branch and $Branch\_B$ is the redundant branch. The decision to start dynamic reconfiguration between branches of a unit i is defined by a function $Unit\_reconf\_need_i$:

$$\begin{aligned} Unit\_reconf\_need_i : \\ Unit_i\_ERRORS \times BRANCH \rightarrow BOOL \end{aligned} \quad (5.5)$$

In the real system, mode transitions may take time and can be interrupted by errors. To unambiguously describe the actual state of a mode managing component at any layer, we define the following attributes for each of them:

1. *prev_target* is the previous mode that a component was in transition to;
2. *last_mode* is the last successfully reached mode;
3. *next_target* is the target mode that a component is currently in transition to.

Based on these attributes, we define the notion of the mode manager state that might be either Stable, Decreasing or Increasing as follows:

Stable_C: a component C is maintaining the last successfully reached mode, i.e., $last\_mode_C = prev\_target_C \wedge next\_target_C = prev\_target_C$;

Increasing_C: a component C is in transition to a next, more advanced mode, i.e., $last\_mode_C = prev\_target_C \wedge prev\_target_C < next\_target_C$;

Decreasing_C: component C stability or a mode transition to the previous target was interrupted (e.g., by error handling) by a new mode request to a more degraded mode, i.e., $next\_target_C < prev\_target_C$.

When a mode transition is completed, the component state becomes stable. Moreover, when the top layered component MM is in a stable state, all the lower layer components, i.e., unit managers, should be stable as well. This can be formulated as an invariant property of the system:

$$Stable_{MM} \Rightarrow Stable_{UM_1} \wedge \dots \wedge Stable_{UM_m}$$

Another main invariant property of the system states that during reconfiguration a component cannot be stable:

$$\forall C \cdot In\_Reconf(C) = TRUE \Rightarrow \neg Stable_C$$

Each unit in the system has not only its local mode but also the status (*Locked* or *Unlocked*). The correspondence between these two notions can be formalised as the following predicate:

$$\begin{aligned} Reconfig(UM_i) = TRUE \vee last\_mode_{UM_i} = Off \vee \\ next\_target_{UM_i} = Off \Rightarrow Status_{UM_i} = Unlocked \end{aligned} \quad (5.6)$$

or, equivalently,

$$\begin{aligned} Status_{UM_i} = Locked \Rightarrow Reconfig(UM_i) = FALSE \wedge \\ last\_mode_{UM_i} \neq Off \wedge next\_target_{UM_i} \neq Off \end{aligned} \quad (5.7)$$

Such a property follows from the definition of the unit status: each unit can have the status *Locked* only if there is no ongoing reconfiguration in this unit and the unit is neither in the non-operational mode *Off*, nor in the transition from or to the mode *Off*.

The discussion above sets general guidelines for defining different layer mode managers and reconfiguration procedures. In Section 5.4.4 we will show how these guidelines can be implemented in a formal specification in Event-B.

5.4.4 Case Study – Attitude and Orbit Control System

The Attitude and Orbit Control System (AOCS) is a typical layered control system. The main function of the system is to control the attitude and the orbit of a satellite. Since the orientation of a satellite may change due to disturbances of the environment, the attitude needs to be continuously monitored and adjusted. The optimal attitude is required to support the needs of payload instruments and to fulfil the mission of the satellite [Depa].

At the top layer of AOCS is *Mode Manager (MM)*. It controls several *Unit Managers (UMs)*, which are responsible for a number of hardware units. AOCS has seven units – four sensors (Star tracker (STR), Sun Sensor (SS), Earth Sensor (ES) and Global Positioning system (GPS)), two actuators (Reaction Wheel (RW) and Thruster (THR)), and one payload instrument (PLI) producing mission measurements. The predefined mode scenario determines the sequence of steps needed to reach the state where PLI is ready to perform its tasks. This sequence includes the following modes:

- **OFF** – the satellite is typically in this mode right after system (re)booting;
- **Standby** – this mode is maintained until separation from the launcher;
- **Safe** – a stable attitude is acquired, which allows the coarse pointing control;
- **Nominal** – the satellite is trying to reach the fine pointing control, which is needed to use the payload instrument;
- **Preparation** – the payload instrument is getting ready;
- **Science** – the payload instrument is ready to perform its tasks. The mission goal is to reach this mode and stay in it as long as needed.

In this section we consider a formal derivation of the AOCS mode logic focusing only on Earth Sensor (ES) and Global Positioning System (GPS). ES is a typical representative of AOCS units with two simple local modes: non-operational, *Off*, and operational, *On*. GPS represents the units with one non-operational mode, *Off*, and two operational modes, *Coarse* and *Fine*. When the global mode is **Safe**, ES is in the operational mode *On*, otherwise it is in the mode *Off*. Similarly, GPS is required to be in the mode *Coarse*, when the global mode is **Nominal**, and in the operational mode *Fine* when the global mode is either **Preparation** or **Science**. It should be in the mode *Off* otherwise (see Table I).

To identify rollbacks according to the method proposed in Section 5.4.2, we have conducted FMEA for all the global modes where either ES or GPS is in an operational mode. We take into account other AOCS units that could influence the rollback. For brevity, in Fig. 5.9 we show an excerpt from FMEA worksheet. It corresponds to the global mode **Nominal** with the failure mode *GPS failure without an available spare*.

Relying on the FMEA results, we define rollback rules for the ES and GPS units. Specifically, if the system is in **Nominal** mode and both branches of GPS failed, the system should perform a rollback to the mode **Safe**. On the

Table 5.1: Correspondence between unit modes and global modes of AOCS

	Mode	OFF	Standby	Safe	Nominal	Preparation	Science
<i>Unit</i>							
<i>ES</i>		Off	Off	On	Off	Off	Off
<i>GPS</i>		Off	Off	Off	Coarse	Fine	Fine

other hand, if the system is in the **Safe** mode and both branches of ES failed, the system should perform a rollback to the mode **OFF**.

We use refinement process to incrementally build the system architecture, i.e., gradually unfold system layers. The refinement process starts from developing the top level mode manager that implements the global mode logic. The mode manager itself can be developed in a stepwise manner by several refinements. Once the mode manager is specified in sufficient details, we start modelling the lower layer managers together with their mode logics. Each unit manager and its logic can be introduced in separate refinement steps. In the last refinement step we introduce modelling of the redundant branches of each unit.

The abstract specification models a generic mode manager. The variables representing the state of an abstract mode manager (*prev_targ*, *last_mode* and *next_targ*) and the detected errors (the variable *error*) are introduced at this step. The event *Run_Mode_Scenario* models autonomous scenario execution of the abstract mode manager and updates the variable *next_targ*. The events *Error_Handling* and *Error_Reset* describe different error handling procedures (with and without rolling back) of the mode manager, while the event *Error_Occurrence* abstractly models the occurrence of failures. The variable *error* is updated by all these events, while the variables representing the state of the mode manager are only changed by the event *Error_Handling*. The event *Advance_Partial* models partial reaching of the target mode, i.e., after the execution of this event the mode manager state is still Increasing. On the other hand, the event *Mode_Reached* implements complete achievement of the target mode, i.e., after the execution of this event the mode manager state is Stable.

The first refinement is an implementation of MM and its mode logic as a specialisation of the abstract model. According to the formalisation guidelines proposed in Section 5.4.3, we define the allowed AOCS mode transitions

Mode	Nominal		
Failure mode	GPS failure without an available spare		
Possible cause	Hardware failure		
Local effects	Sensor reading is out of expected range. Change GPS status		
System effects	Switch to a degraded mode		
Detection	Comparison of received data with the predicted one		
Remedial action	Target mode	Precondition	Action
	Safe	A state transition error in some of the redundant branches in GPS, STR and THR. No state transition error in the redundant branch of RW.	For each unit other than RW, any ongoing unit reconfiguration is aborted. For each branch in each unit other than RW, the status is set to Unlocked, and a state transition to Off is initiated.
		Insufficient usability of a selected redundant branch of GPS, STR or THR. No branch state transition error. No problem on the redundant branch of RW.	
	OFF	A state transition error in the redundant branch of RW.	For each unit, any ongoing unit reconfiguration is aborted. For each branch in each unit the status is set to Unlocked, and a state transition to Off is initiated.
		Insufficient usability of a selected redundant branch of RW. No branch state transition error.	

Figure 5.9: FMEA of the AOCS global mode Nominal.

(Fig. 5.10). We represent them formally by instantiating the relation *Scenario* (1):

$$\begin{aligned}
\text{Scenario} = \{ & OFF \mapsto Standby, Standby \mapsto Safe, \\
& Safe \mapsto Nominal, Nominal \mapsto Preparation, \\
& Preparation \mapsto Science \}
\end{aligned}$$

The execution of the scenario is modelled as a sequence of events. A more detailed description is given in [ITL⁺10c].

To model rollbacks shown in Fig. 5.10, we introduce a function *MM_Error_Handling* that is an instantiation of the function *Error_Handling* (2) given in Section 5.4.3.

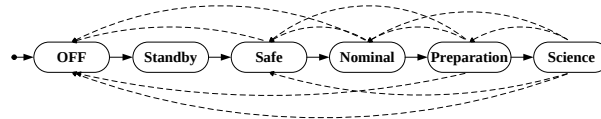


Figure 5.10: AOCS mode transitions.

The second refinement step introduces one of the considered units, Earth Sensor. The next target mode for ES can be set only by a higher level

manager, in our case, by the mode manager. The scenario relation for ES is very simple:

$$ES\_scenario = \{ES\_Off \mapsto ES\_On\}$$

The correspondence relation between the MM and ES modes (shown in Table I) is formalised as follows:

$$\begin{aligned} ES\_mode = \{ & OFF \mapsto ES\_Off, Standby \mapsto ES\_Off, \\ & Safe \mapsto ES\_On, Nominal \mapsto ES\_Off, \\ & Preparation \mapsto ES\_Off, Science \mapsto ES\_Off \} \end{aligned}$$

Essentially, the *ES_mode* relation expresses the required consistency conditions between the modes of MM and ES. It is also a concrete projection of the generic function *Mode_entry_cond* introduced in Section 5.4.3.

In our Event-B model, we describe partial and complete reaching of a new target for ES by the respective events *ES_Advance_Partial* and *ES_Mode_Reached* (Fig 5.11).

<pre> event ES_Advance_Partial // mode is advanced but target mode is not reached yet any p where ES_prev_targ ≠ ES_next_targ p ∈ ES_MODES p ↦ ES_next_targ ∈ ES_graph p ≠ ES_next_targ last_mode ↦ p ∈ ES_mode flag_ES_rec = FALSE then ES_last_mode := p ES_Status := fun ES_status (ES_Reconfig ↦ p ↦ ES_next_targ) end </pre>	<pre> event ES_Mode_Reached // mode is reached where ES_next_targ ≠ ES_prev_targ last_mode ↦ ES_last_mode ∈ ES_mode last_mode ↦ ES_next_targ ∈ ES_mode last_mode ↦ ES_prev_targ ∈ ES_mode flag_ES_rec = FALSE then ES_last_mode := ES_next_targ ES_prev_targ := ES_next_targ ES_Reconfig := FALSE ES_Status := fun ES_status (FALSE ↦ ES_next_targ ↦ ES_next_targ) end </pre>

Figure 5.11: ES mode transition events.

Here the function *fun_ES_status* is used to determine the ES status (*Locked* or *Unlocked*). The function result depends on the given values of the variables *ES_Reconfig*, *ES_last_mode* and *ES_next_targ*. Essentially, the function *fun_ES_status* is defined in such a way that its result (the ES status) always satisfies the following invariant properties of our model:

$$\begin{aligned} ES\_Reconfig &= TRUE \vee ES\_last\_mode = ES\_Off \vee \\ ES\_next\_targ &= ES\_Off \Rightarrow ES\_Status = Unlocked \end{aligned}$$

$$\begin{aligned} ES\_Status = Locked &\Rightarrow ES\_Reconfig = FALSE \wedge \\ ES\_last\_mode &\neq ES\_Off \wedge ES\_next\_targ \neq ES\_Off \end{aligned}$$

The first invariant shows what conditions must be true for the system to have status *Unlocked*. The second property follows from the first one and specifies which conditions are true when the ES status is *Locked*. Please note that these invariants are concrete instantiations of the general properties (4) and (5) given in Section 5.4.3.

The third refinement step implements the GPS unit and its mode logic in a similar way. In the last refinement step we introduce the reconfiguration procedures of ES and GPS units according to the remedial actions described in the FMEA worksheets.

To decide whether reconfiguration is needed, we introduce a function called *fun_ES_rec_need_b* for the ES unit and a similar function *fun_GPS_rec_need_b* for the GPS unit. These functions are concrete instances of the abstract function *Unit_reconf_need_i* (3). The functions return *TRUE*, if reconfiguration is needed, and *FALSE* otherwise.

All the unit errors can be divided on two types: *reconfigurable*, i.e., the unit can recover from them by its reconfiguration, and *non-reconfigurable*, i.e., they cannot be handled by reconfiguration even if the redundant branch is available. The required properties of the above functions are given as axioms in the accompanying CONTEXT model. The **Axiom 1** given below states that, if any reconfigurable error of ES occurs and the current branch is the nominal branch *Branch_A*, then reconfiguration has to be started. **Axiom 2** defines the situation when a reconfigurable error occurs but the current branch is already the redundant branch *Branch_B*. Therefore, unit reconfiguration is not possible and thus is not needed. Finally, **Axiom 3** specifies the case when reconfiguration is not possible due to a non-reconfigurable error.

Axiom 1: $\forall x \cdot x \in ES\_Reconf\_errors \Rightarrow$
 $fun\_ES\_rec\_need\_b(x \mapsto Branch\_A) = TRUE$

Axiom 2: $\forall x \cdot x \in ES\_Reconf\_errors \Rightarrow$
 $fun\_ES\_rec\_need\_b(x \mapsto Branch\_B) = FALSE$

Axiom 3: $\forall x \cdot x \in Any\_Error \setminus ES\_Reconf\_errors \Rightarrow$
 $fun\_ES\_rec\_need\_b(x \mapsto Branch\_A) = FALSE$

The introduced reconfiguration mechanism affects a number of model events. In particular, the events modelling unit failures that may trigger reconfiguration are refined to update the reconfiguration status (modelled as a separate flag variable) of a unit. The event *ES_Timeout_Err* presented in Fig. 5.12 is an example of that. Moreover, separate events starting the reconfiguration process are introduced for each unit as special refined versions

of the abstract event *Error_Reset*. The event *ES_Reconfig_Start* shown in Fig. 5.12 is an example of that for the unit ES. Note that it is enabled only when reconfiguration has been already triggered by some previous unit failure, i.e., when $flag\_ES\_rec = TRUE$.

<pre> event ES_Timeout_Err refines Error_Occurrence where ES_next_targ $\neq$ ES_prev_targ ES_next_targ $\neq$ ES_Off error = No_Error then error := ES_Timeout_Error flag_ES_rec := fun_ES_rec_need_b (ES_Timeout_Error $\mapsto$ ES_branch) end </pre>	<pre> event ES_Reconfig_Start refines Error_Reset where ES_Reconfig = FALSE flag_ES_rec = TRUE then ES_Reconfig := TRUE ES_Status := Unlocked flag_ES_rec := FALSE error := No_Error ES_last_mode := ES_Off ES_prev_targ := ES_Off ES_branch := Branch_B end </pre>

Figure 5.12: Reconfiguration between ES branches.

If the reconfiguration is not possible, i.e., both the nominal and redundant branches failed, the mode manager performs a rollback to such a global mode where the failed component is not needed, i.e., it is in the mode *Off*. The corresponding event is shown in Fig. 5.13.

To perform a rollback transition, all AOCS units have to be moved into appropriate modes according to the correspondence relations between MM and each unit modes (e.g., *ES_mode*, *GPS_mode*, etc). All the variables reflecting the current state of AOCS and all the units (such as *prev_targ*, *last_mode*, *next_targ*, etc) have to be assigned new values. The statuses of units are also changed according to the resulting values of the respective functions (e.g., *fun_ES_status*, etc).

In our Event-B development of AOCS we have applied the general guidelines proposed in Sections 5.4.2 and 5.4.3. We have gradually introduced the concrete modes for different system layers, their interdependencies as well as the dynamic reconfiguration mechanism. The essential mode consistency conditions have been verified in the refinement process. The modelling and verification have been carried out in the Rodin Platform [Rodb]. The respective proof obligations have been discharged using a collection of the provided automated theorem provers with a small number of interactive proofs.

5.4.5 Discussion

There are several well-known problems associated with mode-rich systems including mode confusion and automation surprises [But04, Rus02]. These studies conducted retrospective analysis of mode-rich systems to spot the


```

event MM_Error_Handling refines MM_Error_Handling
any  $m\ u\ gp$ 
where
   $m \in \text{MODES}$ 
   $\text{error} \neq \text{No\_Error}$ 
   $\text{prev\_targ} \mapsto m \in \text{Mode\_Order}\sim$ 
   $m \neq \text{next\_targ} \wedge \text{next\_targ} \neq \text{OFF}$ 
   $m \mapsto u \in \text{ES\_mode} \wedge u \neq \text{ES\_next\_targ}$ 
   $m \mapsto gp \in \text{GPS\_mode} \wedge gp \neq \text{GPS\_next\_targ}$ 
then
   $\text{error} := \text{No\_Error}$ 
   $\text{prev\_targ} := \text{next\_targ}$ 
   $\text{last\_mode} := \text{next\_targ}$ 
   $\text{next\_targ} := m$ 
   $\text{ES\_Status} := \text{fun\_ES\_status}$ 
    ( $\text{ES\_Reconfig} \mapsto \text{ES\_next\_targ} \mapsto u$ )
   $\text{ES\_prev\_targ} := \text{ES\_next\_targ}$ 
   $\text{ES\_last\_mode} := \text{ES\_next\_targ}$ 
   $\text{ES\_next\_targ} := u$ 
   $\text{ES\_Reconfig} := \text{FALSE} \parallel \text{GPS\_Reconfig} := \text{FALSE}$ 
   $\text{GPS\_Status} := \text{fun\_GPS\_status}$ 
    ( $\text{GPS\_Reconfig} \mapsto \text{GPS\_next\_targ} \mapsto gp$ )
   $\text{GPS\_prev\_targ} := \text{GPS\_next\_targ}$ 
   $\text{GPS\_last\_mode} := \text{GPS\_next\_targ}$ 
   $\text{GPS\_next\_targ} := gp$ 
end

```

Figure 5.13: System rollback.

discrepancies between the actual system mode logic and the user mental picture of the mode logic. Most of the approaches have relied on model-checking [But04, HL96, Rus02], while [But96b] is based on theorem proving in PVS. Our approach focuses on designing fully automatic systems and ensuring their mode consistency. Unlike [HL96], in our approach we also emphasize the complex relationships between system fault tolerance and the mode logic. Our approach allows the developers to systematically design the system and formally check mode consistency within the same framework.

In [LT04] the authors have introduced a general formal specification approach to the development of dependable systems with layered architectures. The approach is based on using the exception handling mechanism on each layer of such systems. The exceptions that cannot be handled at a certain layer are propagated to the upper layers. As a result, error recovery has a hierarchical structure. The similar idea is explored in this section as well. In addition, we considered a mode-based mechanism for structuring the system behaviour. This section extends the work presented in [LT04] by incorporating FMEA into the process of designing layered mode-rich systems. Moreover, we consider an interplay between unit reconfiguration and mode consistency.

The contribution of this section is two-fold. Firstly, we have proposed a systematic approach to deriving the fault tolerance part of a mode logic

using FMEA. Secondly, we have formalised the required conditions of mode consistency and demonstrated how to ensure them while developing a system by refinement in Event-B. Verification by theorem proving and stepwise refinement have allowed us to undertake a formal development of a complex control system. Hence our approach shows a good scalability.

5.5 Security

In [HBA09], we proposed an approach for developing access control system using Event-B. The key aspect of our approach is to develop independently the insecure target system and the security authorisation. The resulting systems are then combined together (e.g., using *Parallel composition* [Sil]).

We illustrate the approach by an example for compliance modelling from SAP. There are two different types of business objects: *price specification* and *sale order*, where a sale order depends on a particular price specification. Services on these business objects includes *creating*, *deleting* and *modifying*. Access control policy specifies that users can call a service on a particular business object depending on certain assigned context. Moreover, certain actions on business objects are declared to conflict with each other and should not be carried out by the same user.

Using our approach, we first model the insecure system involving creating, deleting and managing the business objects, i.e., price specifications and sale orders. The model focuses purely on the functional aspect of the systems, without any additional constraints for authorisation. Hence, we omit details of the system concerning only to the authorisation, e.g., information about *areas*, *contexts* and *departments*.

In the second step, we model the authorisation, which involves various users of the systems, together with the fore-mentioned business objects. The authorisation policy includes restriction on users when calling a service on a business object and preventing the same user perform conflict actions on different business objects.

In the final step, we combine the two systems, the insecure system together authorisation system using parallel composition. The result is a secure system, which inherits the functionality from the insecure system and the security from the authorisation system.

Using this modelling approach, we separate the concerns about the functionality and security of the systems. The development of two separate sub-systems are done independently: redundant details are left out of the respective sub-systems.

A condition of using parallel composition is that the states of the two

sub-systems (i.e., the insecure system and the authorisation system) must be disjoint. This is not entirely appropriate for our example of compliance modelling. There are some necessarily overlap details (e.g., the identity of the price specification and sale order business objects) need to be “shared” between the sub-systems. Currently, this is modelled in both systems separately, and as a result, duplicated some information in the combined model. We are investigating the use of an alternative approach such as *pattern* [HFA09a], which allow to combine models with the possibility of sharing part of the state.

Chapter 6

Composition and Reuse

*J.R. Abrial, M. Butler, A. Fuerst, A. Gondal, T. Hoang, A. Iliasov,
L. Laibinis, M. Poppleton, R. Silva, W. Wei*

6.1 Decomposition

Modern software systems are getting more complex each day. The trend would continue as industry witnesses unprecedentedly explosive demands for social and business connectivity. This poses a huge challenge to software modelling in service for rigorous quality assurance. Model decomposition has always been a powerful tool to break complexity by separating concerns of different aspects in software designs. Although there has never been a lack of theoretical research in decomposition, very few platforms provide tool supports for actual software development. Consequently, no clear guidelines are available for model designers. This impedes the application of model decomposition as well as formal methods.

Rodin/Event-B has established itself as a popular formal software development platform with broad industrial adoptions. It currently provides three different decomposition plug-ins that cover supports for both top-down designs (as in software development) and bottom-up designs (as in service composition or orchestration). These plug-ins not only provide tool assistance for deciding how to decompose a software model, but also generate explicit constraints and relations between the decomposed model and the resulting components that must be satisfied in order to guarantee a correct decomposition. However, there are still pitfalls in model decomposition that, if not correctly addressed, may lead to invalid and unusable software de-

sign. These pitfalls are currently not captured by the tools. In general it is very difficult to offer generic solutions to these problems because they largely depend on the nature of each individual software model being developed.

In this section we intend to provide useful insights and guidelines of model decomposition, drawn from our experiences of realistic software modeling in collaboration with industrial partners. These guidelines are not supposed to give a one-for-all solution for all decomposition attempts. Instead, we introduce several stages in decomposition and point out which design problems need to be addressed in each step. We highlight design problems that can be easily overlooked. In the end, we also compare three different decomposition approaches, and discuss their strengths and weakness when applied to different kinds of software systems. Although the guidelines and our experiences are explained for Event-B decomposition, they could serve as general guidelines for decompositions with other formalisms.

6.2 Modules

6.2.1 Modelling Modular Systems in Event-B

Recently the Event-B language and tool support have been extended with a possibility to define modules [ITL⁺10b, Modb] – components containing groups of callable operations. Modules can have their own (external and internal) state and the invariant properties. The important characteristic of modules is that they can be developed separately and then composed with the main system.

Module Structure A module description consists of two parts – *module interface* and *module body*. Let M be a module. The module interface is a separate Event-B component. It allows the user of the module M to invoke its operations and observe the external variables of M without having to inspect the module implementation details. The module interface consists of the module interface description MI and its context $MI\_Context$. The context defines the required constants c and sets s . The interface description consists, respectively, of the external module variables w , the external module invariant $M\_Inv(c, s, w)$, and a collection of module operations, characterised by their pre- and postconditions, as shown in Figure 6.1. The primed variables in the operation postcondition stand for the variable values after operation execution. A module interface must satisfy certain consistency conditions typical for Event-B specifications – operation *feasibility* and preservation of the module invariant.

```

INTERFACE MI =
  SEES MI_Context
  VARIABLES mv
  INVARIANT M_Inv(mv)
  INITIALISATION mv :∈ M_Init
  OPERATIONS
    res ← op1 =
      ANY par
      PRE M_Guard1(par, mv)
      POST M_Post1(par, mv, mv', res')
    END
  ...
END

```

Figure 6.1: Structure of interface component

A module development always starts with the design of an interface. After an interface is formulated, it cannot be altered in any manner. This ensures correct relationships between a module interface and its body, i.e., that the specification of an operation call is recomposable with an operation implementation. A module body is an Event-B machine. It implements each operation described in the module interface by a separate group of events. Additional proof obligations are generated to verify correctness of a module. They guarantee that each event group faithfully satisfies the given pre- and postconditions of the corresponding interface operation.

Importing of a Module When the module **M** is imported into another Event-B machine, this is specified by a special clause **USES** in the importing machine, **N**. As a result, the machine **N** can invoke the operations of **M** as well as read the external variables of **M** listed in the interface **MI**.

To make a module interface generic, in **MI_Context** we can define some abstract constants and sets (types). Such data structures become module parameters that can be instantiated when a module is imported. The concrete values or constraints needed for module instantiation are supplied within the **USES** clause of the importing machine. Alternatively, the module interface can be *extended* with new sets, constants, and the properties that define new data structures and/or constrain the old ones. Such an extension produces a new, more concrete module interface. Via different instantiation of generic parameters the designers can easily accommodate the required variations when developing components with similar functionality. Hence module instantiation provides us with a powerful mechanism for reuse.

We can create several instances of the given module and import them into the same machine. Different instances of a module operate on disjoint state spaces. Identifier prefixes can be supplied in the **USES** clause to distinguish

the variables and the operations of different module instances or those of the importing machine and the imported module. The syntax of USES then becomes as follows:

USES *< module interface >* **with prefix** *< prefix_ >*

Semantics of Module Interface Similarly to a machine component, the semantics of an interface component is defined by the following proof obligations. The module initialisation must establish the module invariant M_Inv :

$$M\_Init(mv) \vdash M\_Inv(mv) \quad (\text{MOD\_INIT})$$

Let us assume $Oper_i$, $i \in 1..N$, is one of module operations. The module invariant M_Inv should be preserved by each operation execution:

$$M\_Inv(mv), Pre_i(pars, mv), Post_i(pars, mv, mv', res') \vdash M\_Inv(mv') \quad (\text{MOD\_INV})$$

where Pre_i and $Post_i$ are respectfully the precondition and the postcondition of $Oper_i$.

Finally, there is a couple of feasibility proof obligations for each $Oper_i$, $i \in 1..N$. Firstly, the operation precondition should be true for at least some of parameter values:

$$M\_Inv(mv) \vdash \exists pars. Pre_i(pars, mv) \quad (\text{MOD\_PARS})$$

Secondly, at least some operation post-state containing the required result must be reachable:

$$M\_Inv(mv), Pre_i(pars, mv) \vdash \exists(mv', res'). Post_i(pars, mv, mv', res') \quad (\text{MOD\_RES})$$

Semantics of an Operation Call A machine importing a module instance operates on the extended state consisting of its own variables v and module variables mv . The module state can be updated in event actions only via operations calls. The semantics of an event containing an operation call is as follows.

Please recall that the semantics of Event-B actions is defined using before-after (BA) predicates [Abr10], which describes a relationship between the system states before and after an execution of an event action. The notion of BA predicate can be easily generalised to formally define model events. For an event e of the form **ANY** lv **WHERE** g **THEN** S **END**, its BA predicate is as follows:

$$BA_e(x, y, x') = \exists lv. g(lv, x, y) \wedge BA_S(x, y, lv, x')$$

Let us consider the model event E that contains a call to the module operation Op with the given arguments $args$, i.e., it is of the form

ANY $lvars$ **WHERE** g **THEN** $S[Op(args)]$ **END**.

The BA predicate of such an event can be then defined as follows:

$$\begin{aligned} BA_E(v, mv, v', mv') = & \exists(lvars, res, new\_mv). g(lvars, v, mv) \wedge \\ & Post(args, mv, new\_mv, res) \wedge \\ & BAS^*(lvars, v, mv, res, v') \wedge (mv' = new\_mv) \end{aligned}$$

where S^* is S with all the occurrences of $Op(args)$ replaced by res . Once this is done, we can rely on the existing proof semantics to verify the invariant preservation, the event simulation and other required properties.

Moreover, we need an additional proof obligations to ensure call correctness by checking that the operation precondition holds at the place of an operation call:

$$g(lv, v, mv), Inv(v, mv), M\_Inv(mv) \vdash Pre(args, mv) \quad (\text{CALL\_CORR})$$

6.2.2 Conclusions

The modularisation extension of Event-B facilitates formal development of complex industrial-size systems by allowing the designers to decompose large specifications into separate components and verify system-level properties at the architectural level. The presented approach is implemented as a plug-in to the RODIN platform. Currently we are working on the further extensions of the modularisation approach including, in particular, a possibility to define indexed modules (classes) as well as broadcasted calls to many module instances.

There are three major approaches to decomposition and modularisation. One is to identify a general theory that, once formally defined, would contribute to the main development. For instance, a model realising a stack-based interpreter could be simplified by considering the stack concept in isolation, creating a general theory of stacks and then reusing it in the main development. Such an approach was investigated in, e.g., [Fit91].

Another form of decomposition is based on splitting a system into a number of parts and then proceeding with independent development of each part. At some point, the model parts are recomposed to construct the overall final model. This decomposition style often relies on the monotonicity of program refinement, although further constraints must be satisfied to ensure the validity of a recomposed model [AL93, Rod05, But09d]

Finally, decomposition may be realised by hierarchical model structuring, where part of an overall system functionality is encapsulated in a self-contained modelling unit embedded into another unit. Such an approach is conceptually close to a procedure (function) call in programming languages. It helps to structure detailed models in a way a system would be realised in software. This is the approach we have taken in this work.

In particular, we have proposed a pragmatic approach to supporting modularisation in Event B. This work was motivated by the formal development conducted by Space Systems Finland [Var09]. The analysis of the development has shown that the lack of modularisation makes the approach unscalable. Yet the top-down development paradigm and automated verification offer an attractive design platform. Our conservative extension of Event B alleviates scalability problem while preserving all the benefits.

The proposed approach to modularisation can be seen as a special case of the "shared variables" type of decomposition by J.-R. Abrial [Rod05]. Abrial aims at enabling decomposition for distributed systems. In our case, the systems under construction are sequential, even though their functionality is distributed among several modules. Our goal was to enable parallel development of several subsystems as well as reuse formally developed modules. Other proposals include the "shared events" style decomposition for distributed systems [But09d] as well as supporting event fusion in Event B [Pop08]. However, all these works offer more general and hence more difficult to implement alternatives for modularisation.

6.3 Reuse via Application of Modelling and Refinement Patterns

This section presents three different approaches for facilitating reuse by application of pre-defined solutions – design and refinement patterns. The developed patterns are integrated into the refinement process of Event-B.

6.3.1 Design Patterns

In [HFA09b], we have proposed an approach for reusing an Event-B development in such a way that the correctness of models are preserved. In our concept, an Event-B development can be seen as a pattern: the initial model is the description of the problem, whereas the refinement chain are the solution to the problem. Our hypothesis here is that by identifying commonly appeared patterns in certain domain-specific problem, one can develop a general single solution to solve this problem, and apply this solution several time.

This will allows us to reduce the effort of formal modelling and proving in Event-B.

Our approach is illustrated in Figure 6.2. On the left-hand side we have a *pattern* development represented by an abstract model (the specification) p_0 and a concrete refinement p_1 (in general, it could be several refinement layers). On the right-hand side, we have a *problem* development carries out over several refinement steps. At certain level of refinement, e.g. m_n , a certain matching with the pattern specification is identified. Given some syntactic constraint about the matching, the pattern refinement can be incorporated with possibility of renaming variables/events, in order to build a refinement m_{n+1} of the problem development.

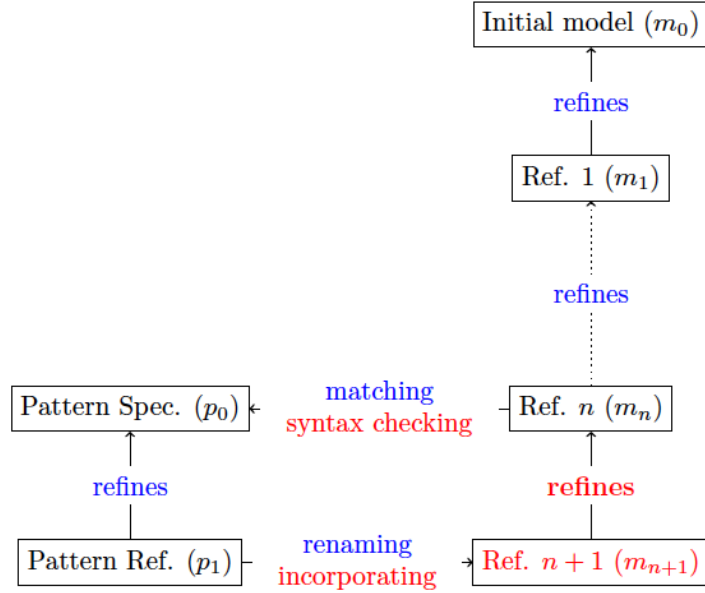


Figure 6.2: Pattern application

Currently, in order to make sure that the generated refinement m_{n+1} is correct-by-construction, we required the following for the matching

- Every variable of the pattern specification has to be matched with some variable in the problem.
- Every event of the pattern has to be match with some event in the problem.

The pattern approach has been illustrated with SAP's mini-pilots and reported in earlier deliverables. Some proposal has been made to incorporate

the approach within the graphical user interface similar to the one used by SAP.

In order to be able to reuse the model more flexible, there is a proposal for instantiating Event-B model. Intuitively, an Event-B development is parameterised by the enclosing carrier sets and constants, which are constrained by axioms. Instantiation technique allows to replace abstract carrier sets and constants by concrete values within the a new context, so that the development can be reused within the new context. In order to make sure that correctness is preserved, we generate proof obligations to ensure that the concrete values satisfying the declared axioms of the instantiated development and also the implicit axioms about carrier sets, i.e. non-empty and maximality. More details can be seen in the proposal on the Event-B wiki [Eve]. After instantiation, the instantiated development can be use as a pattern in order to incorporated in different problem development.

The constraints that we have on applying pattern is strong. We are investigating several situations where the constraints can be relaxed, to increase the applicability of the pattern approach in a wider context. In the most general case, it seems that the two developments, i.e. pattern and problem can be *fused* together, something in between to shared-variable and shared-event composition. The goal for us is to have better methods for reusing, which preserves correctness, but also can be easily applied with some additional tool support.

Two plug-ins has been built separately to support the two techniques. A natural step is to merge the two plug-ins to be able to collect user input, and to perform generic instantiation and pattern at the same time.

6.3.2 Model Transformation Patterns

Formal modelling and verification are valuable for ensuring system dependability. However, often formal development process is perceived as being too complex to be deployed in the industrial engineering process. Hence, there is a clear need for methods that facilitate adopting of formal modelling techniques and increase productivity of their use.

Reliance on patterns – the generic solutions for certain typical problems – facilitates system engineering. Indeed, it allows the developers to document the best practices and reuse previous knowledge.

We adopt a software engineering approach to documenting and reusing Event-B modelling patterns. In this approach, a pattern is technically a model transformation script written in a simplified EMF-oriented programming language EOL [EOL]. Once developed, a script can be executed by a user thus instantiating a pattern and introducing the desired behaviour into

a model. Patterns can be of various specificity, from domain- and project-specific to simple routine automation. It is up to system developers to decide which aspects of modelling to be automated.

The tool support for transformation patterns provides a text editor, a GUI for choosing a pattern for instantiation, and a GUI for user input during execution such as for specifying Event-B elements asked by the pattern. For the development of patterns, the API provides facilities for accessing Event-B and other EMF-based models as trees of EMF elements. During instantiation, a pattern script is executed against the models and Event-B elements specified by a user. More technical details can be found at [pat].

Such an engineering approach to patterns is used in our work on representing the results of Failure Modes and Effects Analysis (FMEA) in Event-B models of control systems [LPT⁺10].

Patterns for Representing FMEA in Event-B FMEA is a widely-used inductive technique for safety analysis [FMEa, Lev95, ?]. We define a set of patterns formalising the requirements derived from FMEA and automate their integration into the formal specification. The application of instantiated patterns automatically transforms a model to represent the results of FMEA in a coherent and complete way.

FMEA patterns for control systems that we identify involve three levels of abstraction:

- On the first level, we model an abstract cyclic behaviour of any control system
- The second level is constructed by applying specific patterns derived from FMEA
- The patterns leave the third level to developers for further refinement of functional behaviour of the system

The control systems are usually cyclic, i.e., at periodic intervals they get input from sensors, process it and output the new values to the actuators. In our specification the sensors and actuators are represented by the corresponding state variables. We follow the systems approach, i.e., model the controller together with its environment – the plant. This allows us to explicitly state the assumptions about environment behaviour. At each cycle the plant assigns the variables modelling the sensor readings. They depend on the physical processes of the plant and the current state of the actuators. In its turn, the controller reads the variables modelling sensors and assigns the variables modelling the actuators. We assume the controller reaction

takes negligible amount of time and hence the controller can react properly on changes of the plant state.

The general specification pattern for modelling a failsafe control system in Event-B is presented in [LPT⁺10]. It represents the overall behaviour of the system as an interleaving between the events modelling the plant and the controller. The abstract model contains certain Event-B elements (variables and events) which the more specific FMEA patterns refine and therefore depend upon.

At the first refinement step we aim at introducing models of system components, error detection procedures as well as error masking and recovery actions. To systematically define failure modes, detection and recovery procedures, for each component, we conduct Failure Modes and Effects Analysis. An example of a FMEA sheet for one of the failure modes of the *door1* component of our example [LPT⁺10] is shown on Table 6.1.

Table 6.1: FMEA table for "out of predicted range" failure mode of one of the sensors

Component	Door1
Failure mode	Door position sensor value is different from the expected range of values
Possible cause	Failure of the position sensor
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the received value with the predicted range of values
Remedial action	Retry. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm

To represent the desired behaviour described in Table 6.1, we map each of the table fields to one or more patterns. We distinguish four groups of patterns: detection, recovery, prediction and invariants. The detection patterns reflect such generic mechanisms for error detection as discrepancy between

the actual and expected component state, sensor reading outside of the feasible range etc. The recovery patterns include retry of actions or computations, switch to redundant components and safe shutdown. The prediction patterns represent the typical solutions for computing estimated states of components, e.g., using the underlying physical system dynamics or timing constraints. Finally, the invariant patterns are usually used in combination with other types of patterns to postulate how a model transformation affects the model invariant. This type contains safety and gluing invariant patterns. The safety invariant patterns define how safety conditions can be introduced into the model. The gluing invariant patterns depict the correspondence between the states of refined and abstract model.

The detection of the mode shown on Table 6.1 can be represented in Event-B by applying the *Expected range pattern* (Snippet 1). The pattern adds the detection events and the variables required to model error detection: expected minimal and maximal values. The pattern ensures that the detection checks added previously by other patterns are preserved (this is informally shown in the angle brackets). The expected range of values used by this pattern must be assigned by some event in the previous control cycle. To ensure that such assignment exists in the model, the *Expected range pattern* instantiates the *Range prediction pattern*. An application of this pattern results in a non-deterministic specification of prediction. It can be further refined to take into account the specific functionality of the system under development.

The remedial action from Table 6.1 can be divided into three actions. The first action retries reading the sensor for a specified number of times (*Retry recovery pattern*). The second action deactivates the faulty component and activates its spare (*Component redundancy recovery pattern*). The third action is enabled when the spare component has also failed. It switches the system from the operational state to the non-operational one (*Safe stop recovery pattern*). The system effect can be represented as a safety property (*Safety invariant pattern*).

To establish refinement between the model created using patterns and the abstract model, we use the *Gluing invariant pattern*, which links the sensor failure with the component failure (Snippet 2). In both snippets, elements in square brackets represent the generic placeholders that will require user input during pattern instantiation (such as strings for names, or existing Event-B elements).

In the second refinement step we introduce the detailed specification of the normal control logic. This refinement step leads to refining the event *Normal_Operation* into a group of events that model the actual control algorithm.

Snippet 1 Expected range pattern

variables

[component]_[sensor]_[error]

[component]_fail

[sensor]_exp_min

[sensor]_exp_max

invariant

inv1 : [component]_[sensor]_[error] $\in$ *BOOL*

inv2 : [component]_fail $\in$ *BOOL*

inv3 : [sensor]_exp_min $\in$ *NAT*

inv4 : [sensor]_exp_max $\in$ *NAT*

events

event Detection_[component]_checks $\hat{=}$

begin

where

grd1 : flag = *DET*

grd2 : Stop = *FALSE*

then

act1 : [component]_[sensor]_[error] := bool(
[sensor]_value < [sensor]_exp_min $\vee$ [sensor]_value >
[sensor]_exp_max)

actN : < other checks >

end

event Detection_[component] $\hat{=}$

begin

where

grd1 : flag = *DET*

grd2 : Stop = *FALSE*

then

act1 : [component]_fail := bool([component]_[sensor]_[error] $\vee$
< other check statuses >)

end

end

Snippet 2 Gluing invariant pattern

```
[component]_fail
```

```

inv1 :  $flag \neq DET \Rightarrow (Failure = TRUE \Leftrightarrow$ 
            $[component]_{fail} = TRUE \vee \langle other\ component\ failures \rangle)$ 
inv2 :  $flag \neq CONT \Rightarrow ([component]_{fail} = TRUE \Leftrightarrow$ 
            $[component]_{[sensor]_{[error]}} = TRUE \vee \langle other\ sensor\ errors \rangle$ 
        )
end

```

The model transformation pattern tool was successfully used for the implementation of the proposed patterns, and showed to be a pragmatic solution to reuse of established modelling practices.

6.3.3 Guided Formal Development: Patterns for Modelling and Refinement

Introduction

In general, a refinement pattern is a generic model transformer. Essentially it consists of three parts. The first part is the pattern applicability conditions, i.e., the syntactic and semantic conditions that should be fulfilled by the model to be eligible for a refinement pattern application. The second part contains definition of syntactic manipulations over the model to be transformed. Finally, the third part consists of the proof obligations that should be discharged to verify that the performed model transformation is indeed a refinement step.

model transformation steps can be represented by a certain combination of refinement patterns, and therefore can also be seen as refinement patterns per se. A possibility to compose patterns significantly improves scalability of formal modelling. Moreover, reducing execution of a refinement step to a number of syntactic manipulations over a model provides a basis for automation. Finally, our approach can potentially support reuse of not only models but also proofs. Indeed, by proving that an application of a generic pattern produces a valid refinement of a generic model, we at the same time verify the correctness of such a transformation for any of its instances. This might significantly reduce or even avoid proving activity in a concrete development.

The theoretical work on defining refinement patterns presented here established a basis for building a prototype tool for automating refinement process in Event B [Ili08a]. The tool has been developed as a plug-in for the RODIN platform. We believe that, by creating a large library of refinement patterns and providing automated tool support for pattern matching and instantiation, we will make formal modelling and verification more accessible for software engineers and hence facilitate integration of formal methods into software engineering practice.

Refinement Patterns

Event-B Models as Syntactic Objects To define refinement patterns, we now consider an Event B model as a syntactic mathematical object. For brevity, we omit representations of some model elements here, even though they are supported in our tool implementation. The subset of Event B models used here can be described by the following data structure:

model	::	<i>var</i> : VAR*	event	::	<i>name</i> : EVENT	action	::	<i>var</i> : VAR
		<i>inv</i> : PRED*			<i>param</i> : PARAM*			<i>style</i> : STYLE
		<i>evt</i> : event*			<i>guards</i> : PRED*			<i>expr</i> : EXPR
					<i>actions</i> : action*			

Here VAR, PRED, EXPR, EVENT, PARAM are the carrier sets reserved correspondingly for model variables, predicates, expressions, event names and parameters. An event is represented by a tuple containing the event name, (a list of) its parameters, guards, and actions. The reserved event name `init` denotes the initialisation event. An action, in its turn, is a tuple containing a variable, an action style and an expression, where an action style denotes one of the assignment types : i.e., $STYLE = \{:=, :\in, :|\}$.

Sub-elements of a model element can be accessed by using the dot operator: *act.style* is the style of an action *act*. Instances of the models, events and actions are constructed using a special notation $\langle a_1 \mid \dots \mid a_n \rangle$. The following example shows how an Event B model is represented in our notation:

```

MACHINE m0

VARIABLES x                                 $\langle \langle x \rangle \mid$ 
INVARIANT  $x \in \mathbb{Z}$                          $\langle "x \in \mathbb{Z}" \rangle \mid$ 
INITIALISATION  $x := 0$                        $\langle \langle \text{init} \mid - \mid - \mid \langle x := "0" \rangle \rangle,$ 
EVENTS
    count = BEGIN  $x := x + 1$  END           $\langle \langle \text{count} \mid - \mid - \mid \langle x := "x + 1" \rangle \rangle \rangle$ 

```

In the example, x is an element of VAR, **init** and **count** are event names from EVENT, " $x \in \mathbb{Z}$ " is a predicate, and " 0 ", " $x+1$ " are model expressions.

Now we have set a scene for a formal definition of refinement patterns that aim at automating refinement process in general and Event B in particular.

Definition 1 Let S be a set of all well-formed models defined according to the syntax of Event B. Then a transformation rule T is a function computing a new model for a given input model:

$$T : S \times C \text{ functional } S$$

where C contains a set of all possible configurations (i.e., additional parameters) of a transformation rule.

Note that T is defined as a partial function, i.e., it produces a new model only for some acceptable input models s and configurations c , i.e., when $(s, c) \in \text{dom}(T)$.

Definition 2 A refinement pattern is a transformation rule $P : S \times C \text{ functional } S$ that constructs a model refinement for any acceptable input model and configuration:

$$\forall s, c. (s, c) \in \text{dom}(P) \Rightarrow s \sqsubseteq P(s, c)$$

where $\sqsubseteq$ denotes a refinement relation.

We rely on the Event-B proof theory when demonstrating that a transformation rule is indeed a refinement pattern.

The Language of Transformations

We propose a special language to construct transformation rules. The proposed language contains basic transformation rules as well as the constructs allowing to compose complex rules from simpler ones. For instance, a refinement pattern is usually composed from several basic transformation rules. These rules themselves might not be refinement patterns. However, by attaching to them additional proof obligations, we can verify that their composition becomes a refinement pattern.

The structure of the basic rules reflects the way a transformation rule or a refinement pattern is applied. First, rule applicability for a given input model and configuration parameters is checked. The applicability condition to be checked can contain both syntactic and semantic constraints on input models and configurations. Mathematically, for a transformation rule T , its applicability condition corresponds to $\text{dom}(T)$. Then, the input model s for the given configuration c is syntactically transformed into the output model calculated as function application $T(s, c)$. Finally, in case of a refinement pattern, the result $T(s, c)$ should be demonstrated to be a refinement of the input model s , i.e., $s \sqsubseteq T(s, c)$. The last expression, using the proof theory of Event B, can be simplified to specific proof obligations on model elements to be verified.

A basic rule has the following general form:

```

rule name(c)
  context  $Q(c, s)$ 
  effect  $E(c, s)$ 
  proof obligation  $PO_1(c, s)$ 
  ...
  proof obligation  $PO_n(c, s)$ 

```

Here *name* and *c* are correspondingly the rule name and the list of its parameters. The predicate $Q(c, s)$ defines the rule application context (applicability conditions), where s is the model being transformed. The effect function $E(c, s)$ computes a new model from a current model s and parameters c . The proof obligation part contains a list of theorems to be proved to establish that the rule is a (part of) refinement pattern and not just a transformation rule. From now on, we write **context**(r), **effect**(r) and **proof_obligations**(r) to refer to the context, effect computation function, and collection of proof obligations of a rule r .

As an example, let us consider two primitive rules for the Event-B method. The first transformation adds one or more new variables:

rule *newvar*(*vv*)

context $vv \cap s.var = \emptyset$

effect $\langle s.var \cup vv \mid s.inv \mid s.evt \rangle$

proof_obligation $\forall v \in vv \cdot (\exists a \cdot a \in s.init.action \wedge v \in a.var)$

The rule applicability condition requires that the new variables have fresh names for the input model. The effect function simply adds the new variables to the model structure. The rule also has a single proof obligation requiring that the variable(s) is assigned in the initialisation action. Such an action would have to be added by some other basic rule for the same refinement step.

Another example is the rule for adding new model invariant(s).

rule *newinv*(*ii*)

context $ii \subseteq \text{PRED} \wedge \forall i \in ii \cdot FV(ii) \subseteq s.var$

effect $\langle s.var \mid inv \cup ii \mid evt \rangle$

proof obligation

$\forall e, v, v' \cdot e \in s.evt \wedge$

$Inv(v) \wedge Guards_e(v) \wedge BA_e(v, v') \Rightarrow Inv(v')$

proof_obligation $\exists v \cdot Inv(v)$

Here $FV(x)$ is set of free variables in x , Inv stands for $(\bigwedge_{i \in s.inv \cup ii} i)$, $Guards_e$ is defined as $(\bigwedge_{g \in e.guards} g)$ and BA_e is the before-after predicate. Both proof obligations are taken directly from the Event-B semantics (i.e., the corresponding proof obligation rules). The first obligation requires to show that the new invariant is preserved by all model events, while the second one checks feasibility of such an addition by asking to prove that the new invariant is not contradictory. This example illustrates how the underlying Event B semantics is used to derive proof obligations for refinement patterns.

The table below lists the basic rules for the chosen subset of Event B. There are two classes of rules – for adding new elements and for removing existing ones. All the rules implicitly take an additional argument – the model being transformed. A double-character parameter name means that a rule accepts a set of elements, e.g., *newgrd*(e, gg) adds all the guards from a given set gg to an event e .

$p(c)$	$=$	$basic(c)$	<i>primitive rule</i>
		$p; q$	<i>sequential composition</i>
		$p q$	<i>parallel composition</i>
		if $Q(c, s)$ then p end	<i>conditional rule</i>
		conf $i : Q(i, c, s)$ do $p(i \cup c)$ end	<i>parameterised rule</i>
		par $i : Q(i, c, s)$ do $p(i \cup c)$ end	<i>generalised parallel composition</i>

Figure 6.3: The language of transformation rules

rule $newvar(vv)$	rule $delvar(vv)$
rule $newinv(ii)$	rule $delinv(ii)$
rule $newevt(ee)$	rule $delevt(ee)$
rule $newgrd(e, gg)$	rule $delgrd(e, gg)$
rule $newact(e, aa)$	rule $delact(e, aa)$
rule $newactexp(e, a, p)$	

To construct more complex transformations, we introduce a number of composition operators into our language. They include the sequential, $p; q$, and parallel, $p||q$, composition constructs. In addition, there is the conditional rule construct, **if** c **then** p **end**, as well as a construct allowing us to introduce additional rule parameters – **conf** $i : Q$ **do** $p(i)$ **end**. Finally, to handle rule repetitions, generalised parallel composition is introduced in the form of a loop construct: **par** $c : Q$ **do** $p(c)$ **end**. The language summary is given in Figure 6.3.

Examples

In this section we present a couple of simple examples of refinement patterns constructed using the proposed language.

New Variable A refinement step adding a new variable can be accomplished in three steps. First, the new variable is added to the list of model variables. Second, the typing invariant is added to the model. Finally, an initialisation action is provided for the variable. The following refinement pattern adds a new variable declared to be a natural number and initialised with zero:

```

conf  $v : \neg (v \in s.var)$  do
   $newvar(\{v\});$ 
   $(newinv(\{v \in \mathbb{N}\}, s) \parallel newact(\mathbf{init}, \{\langle v := "0" \rangle\}))$ 
end

```

The only pattern parameter (apart from the implicit input s) is some fresh name for the new model variable.

A pattern application example is given below. The left-hand side model is an input model and the right-hand side is the refined version constructed by the pattern. The example assumes that variable name q is chosen for parameter v .

<pre> MACHINE $m0$ VARIABLES x INVARIANT $x \in \mathbb{Z}$ INITIALISATION $x := 0$ EVENTS $count = \mathbf{BEGIN} \ x := x + 1 \ \mathbf{END}$ </pre>	<pre> MACHINE $m1$ VARIABLES x, q INVARIANT $x \in \mathbb{Z} \wedge q \in \mathbb{N}$ INITIALISATION $x := 0 \parallel q := 0$ EVENTS $count = \mathbf{BEGIN} \ x := x + 1 \ \mathbf{END}$ </pre>

A more general (and also useful) pattern version could accept a typing predicate and an initialisation action as additional pattern parameters.

Action Split In Event B, an abstract event may be refined into a choice between two or more concrete events, each of which must be a refinement of the abstract event. A simple case of such refinement is implemented by the refinement pattern below. The pattern creates a copy of an abstract event and adds a new guard and its negation to the original and new events. The guard expression is supplied as a pattern parameter.

```

conf  $e, en : e \in s.evt \wedge \neg (en \in s.evt)$  do
   $newevt(en, s);$ 
   $newgrd(en, e.guard) \parallel$ 
   $newact(en, e.action);$ 
  conf  $g : g \in \text{PRED} \wedge FV(g) \subseteq s.var$ 
    do  $newgrd(e, g) \parallel newgrd(en, \neg g)$  end
end

```

The pattern configuration requires three parameters. Parameter e refers to the event to be refined from the input model s , en is some fresh event name, and g is a predicate on the model variables.

The pattern is applicable to models with at least one event. The result is a model with an additional event and a constrained guard of the original event. As an input model for this model we use the model from the previous example.

```

MACHINE m1

VARIABLES  $x$ 

INVARIANT  $x \in \mathbb{Z}$ 

INITIALISATION  $x := 0$ 

EVENTS

  count = WHEN  $x \bmod 2 = 0$  THEN  $x := x + 1$  END

  inc    = WHEN  $\neg(x \bmod 2 = 0)$  THEN  $x := x + 1$  END

```

Here, the pattern parameters are instantiated as follows: e as *count*, en as *inc*, and x as $x \bmod 2 = 0$.

Pattern Composition

In the previous section we defined the notion of a basic transformation rule as a combination of the applicability conditions, transformation (effect) function, and refinement proof obligations. Moreover, In Figure 1, we also introduced various composition constructs for creating complex transformation rules. In this section we will show how we can inductively define the applicability conditions, effect, and proof obligations for composed rules.

Rule Applicability Conditions For a basic rule, the rule applicability condition is defined in its **context** clause. For more complex rules constructed using the proposed language of transformation rules, rule applicability is derived inductively according to the following definition:

$$\begin{aligned}
\mathbf{app}(\mathit{basic})(c, s) &= \mathbf{context}(\mathit{basic})(c, s) \\
\mathbf{app}(p; q)(c, s) &= \mathbf{app}(p)(c, s) \wedge \mathbf{app}(q)(c, \mathbf{eff}(p)(c, s)) \\
\mathbf{app}(p \parallel q)(c, s) &= \mathbf{app}(p)(c, s) \wedge \mathbf{app}(q)(c, s) \wedge \\
&\quad \mathit{inter}(\mathbf{scope}(p), \mathbf{scope}(q)) = \emptyset \\
\mathbf{app}(\mathbf{if } G(c, s) \mathbf{ then } p \mathbf{ end})(c, s) &= G(c, s) \Rightarrow \mathbf{app}(p)(c, s) \\
\mathbf{app}(\mathbf{conf } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s) &= \forall i \cdot Q(i, c, s) \Rightarrow \mathbf{app}(p(i))(c, s) \\
\mathbf{app}(\mathbf{par } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s) &= \mathbf{app}(\mathbf{conf } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s) \wedge \\
&\quad \forall i, j \cdot Q(i, c, s) \wedge Q(j, c, s) \wedge i \neq j \Rightarrow \\
&\quad \mathit{inter}(\mathbf{scope}(p(i)), \mathbf{scope}(p(j))) = \emptyset
\end{aligned}$$

The consistency requirements for the sequential composition, conditional and parameterised rules are quite standard. Two rules can be applied in parallel if they are working on disjoint scopes. For instance, a rule transforming an event (e.g., adding a new guard) cannot be composed with another rule transforming the same event. A similar requirement is formulated for the loop rule, since it is realised as generalised parallel composition.

The rule scopes are calculated by using the predefined function **scope**, which returns a pair of lists, containing the model elements that the rule updates or depends on. Intersection of rule scopes is computed as an intersection of the elements updated by the transformations and the pair-wise intersection of elements updated by one rule and depended on by another:

$$\mathit{inter}((r_1, w_1), (r_2, w_2)) = (w_1 \cap w_2) \cup (r_1 \cap w_2) \cup (r_2 \cap w_1)$$

Effect of Pattern Application Once the rule applicability conditions are met, an output model can be syntactically constructed in a compositional way. For a basic rule, the effect function is directly applied to transform an input model. For more complex rules, a new model is constructed according to an inductive definition of the function **eff** given below.

$$\begin{aligned}
\mathbf{eff}(\mathit{basic})(c, s) &= \mathbf{effect}(\mathit{basic})(c, s) \\
\mathbf{eff}(p; q)(c, s) &= \mathbf{eff}(q)(c, \mathbf{eff}(p)(c, s)) \\
\mathbf{eff}(p \parallel q)(c, s) &= \mathbf{eff}(q)(c, \mathbf{eff}(p)(c, s)), \text{ or} \\
&= \mathbf{eff}(p)(c, \mathbf{eff}(q)(c, s)) \\
\mathbf{eff}(\mathbf{if } G(c, s) \mathbf{ then } p \mathbf{ end})(c, s) &= \mathbf{eff}(p)(c, s), \text{ if } G(c, s) \\
&= s, \text{ otherwise} \\
\mathbf{eff}(\mathbf{conf } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s) &= \mathbf{eff}(p(i))(c, s), \text{ if } Q(i, c, s) \\
&= s, \text{ otherwise} \\
\mathbf{eff}(\mathbf{par } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s) &= \mathbf{eff}(\|i \in Q(i, c, s) \cdot p(i)\|)(c, s), \\
&\quad \text{if } \exists i, c, s \cdot Q(i, c, s) \\
&= s, \text{ otherwise}
\end{aligned}$$

As expected, the result of sequential composition of two rules is computed by applying the second rule to the result of the first rule. For parallel composition, the result is computed in the same manner but the order of the rules should not affect the overall result. The resulting model of the loop construct is computed as generalised parallel composition of an indexed family of transformation rules. The last three cases depend on some additional application conditions (i.e., $G(c, s)$ or $Q(i, c, s)$). If these conditions are not true, rule application leaves the input model unchanged.

The rule application procedure based on the presented definition can be easily automated. The only interesting detail is in providing input values for the rule parameters. In our tool implementation for the Event-B method, briefly covered later, the user is requested to provide the parameter values during rule instantiation, while appropriate contextual hints and descriptions are provided by the tool.

Pattern Proof Obligations To demonstrate that a rule is a refinement pattern, we have to discharge all the proof obligations of individual basic rules occurring in the rule body. These proof obligations cannot be discharged without considering the context produced by the neighbour rules. The following inductive definition shows how the list of proof obligations is built for a particular refinement pattern. The context information for each proof obligation is accumulated, while traversing the structure of a pattern, as a set of additional hypotheses that can be then used in automated proofs.

$$\begin{aligned}
\mathbf{po}(\Gamma, \mathit{basic})(c, s) &= \{\Gamma \models \mathbf{proof\_obligations}(\mathit{basic})\} \\
\mathbf{po}(\Gamma, p; q)(c, s) &= \mathbf{po}(\Gamma \cup \{s' = \mathbf{eff}(p; q)(c, s)\}, p(c, s')) \cup \\
&\quad \mathbf{po}(\Gamma \cup \{s' = \mathbf{eff}(p; q)(c, s)\}, q(c, s')) \\
\mathbf{po}(\Gamma, p \parallel q)(c, s) &= \mathbf{po}(\Gamma, p) \cup \mathbf{po}(\Gamma, q) \\
\mathbf{po}(\Gamma, \mathbf{if } G(c, s) \mathbf{ then } p \mathbf{ end})(c, s) &= \mathbf{po}(\Gamma \cup \{G(c, s)\}, p) \\
\mathbf{po}(\Gamma, \mathbf{conf } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s) &= \bigcup i \in Q(i, c, s) \cdot \mathbf{po}(\Gamma \cup \{Q(i, c, s)\}, p(i)) \\
\mathbf{po}(\Gamma, \mathbf{par } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s) &= \mathbf{po}(\Gamma, \mathbf{conf } i : Q(i, c, s) \mathbf{ do } p(i) \mathbf{ end})(c, s)
\end{aligned}$$

Here Γ is a set of accumulated hypothesis containing pattern parameters c and the initial model s as free variables. For each basic rule, we formulate a theorem whose right-hand side is a list of the rule proof obligations and the left-hand side is a set of hypotheses containing the knowledge about the context in which the rule is applied.

Assertions The described procedure for building a list of proof obligations tries to include every possible fact as a proof obligation hypothesis. This can be a problem for larger patterns as the size of a list of accumulated hypotheses makes a proof obligation intractable. To rectify the problem, we allow a modeller to manually add fitting hypotheses, called assertions, that can be inferred from the context they appear in. An assertion would be typically simple enough to be discharged automatically by a theorem prover. At the same time, it can be used to assist in demonstrating the proof obligations of the rule immediately following the assertion.

An assertion is written as $\mathbf{assert}(A(c, s))$ and is delimited from the neighboring rules by semicolons. An assertion has no effect on rule instantiation and application. The following additional cases of the po definition are used to generate additional proof obligations for assertions as well as insert an asserted knowledge into the set of collected hypotheses of a refinement pattern.

$$\begin{aligned}
\mathbf{po}(\Gamma, p; \mathbf{assert}(A(c, s)))(c, s) &= \Gamma \cup \{s' = \mathbf{eff}(p)(c, s)\} \models A(c, s') \\
\mathbf{po}(\Gamma, \mathbf{assert}(A(c, s)); p)(c, s) &= \mathbf{po}(\Gamma \cup \{A(c, s)\}, p)(c, s)
\end{aligned}$$

Triple Modular Redundancy Pattern

Triple Modular Redundancy (TMR) [LV62] is a fault-tolerance mechanism in which the results of executing three identical components are processed

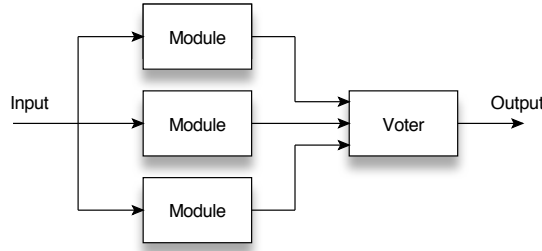


Figure 6.4: TMR Arrangement

by a voting element to produce a single output that takes the majority view. This mechanism is schematically shown in Figure 6.4.

The purpose of the mechanism is to mask a single component failure. In this section we will demonstrate how to generalize a refinement step introducing the TMR arrangement into a model as a refinement pattern.

Before creating our new pattern, we have to decide on its applicability conditions. First, our input model should have a variable representing the output of the component for which TMR will be introduced. Moreover, it should have an event that models the behaviour of a component by non-deterministically updating this variable. Non-determinism is used here to model unpredictable (possibly faulty) results produced by the component. We do not make any assumptions about the variable type. Furthermore, the event can contain some additional actions on other variables. Finally, our input model should also contain an event that handles the component failure.

In the refined model, we replace the single abstract component with three similar components. The outputs of the new components are modelled by fresh variables. The variable types and initialisation of these variables are simply copied from their abstract counterpart in the input specification.

The TMR pattern that we define uses a number of configuration parameters, as shown below. The parameter s identifies a variable modelling the output of a component; u is an event updating the variable s (in addition to possible update of other variables); zz is an event handling a failure of the component modelled by u ; finally, a is an action from u updating the variable s .

```

conf  $s, u, zz, a$  :
   $s \in var \wedge u \in evt \wedge zz \in evt \wedge u \neq zz \wedge$ 
   $a \in u.actions \wedge a.style \neq (:=) \wedge \{s\} = a.var$ 
do
  conf  $ph, s_1, s_2, s_3, r_1, r_2, r_3$  :
     $\{s_1, s_2, s_3, r_1, r_2, r_3, ph\} \subseteq (VAR - var) \wedge$ 
     $part(\{\{s_1\}, \{s_2\}, \{s_3\}, \{r_1\}, \{r_2\}, \{r_3\}, \{ph\}\})$ 
  do
     $variables; events; voter; abort; invariant$ 
  end
end

```

As a result of pattern application, the new variables ph , s_i and r_i are introduced into the refined model. The variable ph keeps track of the current phase in the TMR implementation, i.e., reading from the new components, voting on them, or delivering the final result; the variables s_i , $i = 1..3$, are used to record the outputs produced by the components; finally, the flags r_i reflect availability of new outputs in the respective output variables s_i .

The pattern consists of four major parts: the rules declaring the types and initialisation of new variables of the refined model; the definition of new events; the refinement rules for transforming a single abstract event representing the functionality of a sole component into the voter event; and, finally, the addition of an invariant characterising the behaviour of a TMR block. The condition using the operator *part* simply states that its arguments are disjoint sets.

```

 $variables :=$ 
   $(newinv("ph \in BOOL"); newini(\langle ph \mid := \mid "FALSE" \rangle)) \parallel$ 
   $(newinv("s_1 \in s.type"); newini(\langle s_1 \mid init(s).style \mid init(s).expr \rangle)) \parallel$ 
   $(newinv("r_1 \in BOOL"); newini(\langle r_1 \mid := \mid "FALSE" \rangle))$ 
  ...

```

Each new variable definition should come with a typing invariant and an initialisation action. These are normally grouped together so that the related proof obligation rules would work with a smaller context. For the sake of brevity, we omit showing here the rules defining the types and initialisation for the variables s_2, s_3 and r_2, r_3 (the omitted part is indicated by ...).

The shortcut notation $newini(a)$ used in the pattern description stands for declaration of the initialisation action: $newini(a) := newact(\mathbf{init}, a)$. The shortcut $init(v)$ refers to an action of the initialisation event assigning to the variable v .

The refined model specifies the behavior of three components of TMR (we call them replicated components) as copies of the behaviour of the component specified in the input model. Since we assumed that a component is represented by a single event, the replicated components are created by adding three new events into the refined model in the following way.

The guard of the event modelling behaviour of a replicated component essentially coincides with the guard of an abstract component. However, it also contains an extra conjunct ensuring that the event is executed before passing control to the voter. The event actions essentially copy the corresponding actions of the abstract component (given as the pattern parameter a). The only difference is that each replicated event records the result into a separate variable s_i (for the copy i) instead of the abstract variable s . In addition, a component copy also assigns to r_i to indicate the availability of result in s_i .

```

events :=
conf  $u_1, u_2, u_3$  :
     $\{u_1, u_2, u_3\} \subset \text{EVENT} \setminus s.evt \wedge part(\{\{u_1\}, \{u_2\}, \{u_3\}\})$ 
do
     $copy_1 \parallel copy_2 \parallel copy_3$ 
end

```

The above creates three component copies, each constructed according to the following rule.

```

copy1 :=
    newevt( $\langle u_1 \mid - \mid \{ "r_1 = FALSE" \} \cup u.guards \mid$ 
         $\langle s_1 \mid a.style \mid a.expression \rangle, \langle r_1 \mid := \mid "TRUE" \rangle, \langle ph \mid := \mid "FALSE" \rangle$ )
    ...

```

The above rule $\langle s_1 \mid a.style \mid a.expression \rangle$ constructs an action from the abstract action a in such a way that it would have the same effect but update the new variable s_1 . Here $a.style$ is one of non-deterministic assignment styles.

The voter event is simply a refined version of the event modelling the abstract component. Whereas the abstracted version was computing results

itself, its refined counterpart votes on the results of component copies. The voter is enabled once all the components have produced a result (which is ensured by the first guard in the rule below). The final result is computed according to a simple majority voting protocol. The event parameter rr is set to the voting outcome in the second guard.

```

voter :=
  newpar(u, "rr");
  newgrd(u, "r1 = TRUE ∧ r2 = TRUE ∧ r3 = TRUE");
  newgrd(u, "(s1 = s2 ∨ s1 = s3 ∧ rr = s1) ∨ (s2 = s1 ∨ s2 = s3 ∧ rr = s2)");
  (delact(u, a); newact(u, ⟨s :=| "rr"⟩);
  (newact(u, ⟨r1 :=| "FALSE"⟩) ||
   newact(u, ⟨r2 :=| "FALSE"⟩) ||
   newact(u, ⟨r3 :=| "FALSE"⟩));
  newact(u, ⟨ph :=| "TRUE"⟩)

```

As a result, the abstract action a of the component is replaced by a deterministic assignment (to the same variable s) of the result of the winning component. The flags r_i and ph are reset in preparation for the next iteration.

In case all the component copies disagree, no final result may be computed. This corresponds to an *abort* event of the abstract specification. The refined model simply constraints the guard of the event so it only gets enabled in the situations when the voting has failed.

```

abort :=
  newgrd(zz, "r1 = TRUE ∧ r2 = TRUE ∧ r3 = TRUE");
  newgrd(zz, "s1 ≠ s2 ∧ s2 ≠ s3 ∧ s1 ≠ s3");

```

Finally, a new invariant is added to the refined model to characterise the state of the refined system after voting is completed. It summarises the cases when the majority voting on component results succeeds.

```

invariants :=
  newinv("ph = TRUE ∧ (s1 = s2 ∨ s2 = s3) ⇒ s = s1");
  newinv("ph = TRUE ∧ s2 = s3 ⇒ s = s2");

```

Application of the pattern to a fairly simple abstract model (containing only two events and two variables) saves a user from analysing and discharging 14 proof obligations, three of which would have to be done manually in an interactive theorem prover. For larger models or more elaborated patterns, the benefits are even greater.

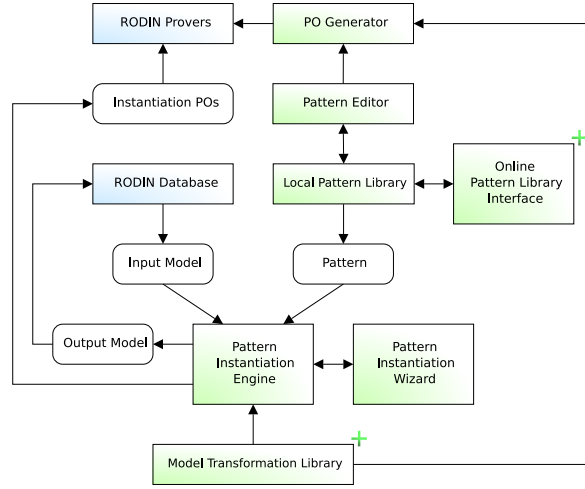


Figure 6.5: The Event-B refinement patterns tool architecture.

Tool for Refinement Automation

A proof of concept implementation of the pattern tool for the Event B method has been realised as a plug-in to the RODIN Platform. The plug-in seamlessly integrates with the RODIN Platform interface so that a user does not have to switch between different tools and environments while applying patterns in an Event B development. The plug-in relies on two major RODIN Platform components: the Platform database, which stores models, proof obligations and proofs constituting a development; and the prover which is a collection of automated theorem provers supplemented by the interactive prover.

The overall tool architecture is presented in Figure 6.5. The core of the tool is the *pattern instantiation engine*. The engine uses an input model, imported from the Platform database, and a pattern, from the pattern library, to produce a model refinement. The engine implements only the core pattern language: the sequential and parallel composition, and *forall* construct. The method-specific model transformations (in this case, Event-B model transformations) are imported from the *model transformation library*.

The process of a pattern instantiation is controlled by the *pattern instantiation wizard*. The wizard is an interactive tool which inputs pattern configuration from a user. It validates user input and provides hints on selecting configuration values. Pattern configuration is constructed in a succession of steps: the values entered at a previous step influence the restrictions imposed on the values of a current step configuration.

The result of a successful pattern instantiation is a new model and, possibly, a set of instantiation proof obligations - additional conditions that must

be verified every time when a pattern is applied. The output model is added to a current development as a refinement of the input model and is saved in the Platform database. The instantiation proof obligations are saved in an Event B *context* file. The RODIN platform builder automatically validates and passes them to the Platform prover.

The tool is equipped with a *pattern editor*. The current version (0.1.7)[Ili08a] uses the XML notation and an XML editor to construct patterns. The next release is expected to employ a more user-friendly visual editor. The available refinement patterns are stored in the *local pattern library*. Patterns in the library are organised in a catalogue tree, according to the categories stated in pattern specifications. A user can browse through the library catalogue using a graphical dialogue. This dialogue is used to select a pattern for instantiation or editing.

When constructing a pattern, a user may wish to generate the set of pattern correctness proof obligations. Proof obligations are constructed by the proof obligation generator component. The component combines a pattern declaration and the definitions of the used model transformations to generate a complete list of proof obligations, based on the rules given in Section 4.3. The result is a new context file populated with theorems corresponding to the pattern proof obligations. The standard Platform facilities are used to analyse and discharge the theorems.

We believe it is important to facilitate pattern exchange and thus the tool includes a component for interfacing with an on-line pattern library. The on-line pattern library and the model transformation library are the two main extension points of the tool. The pattern specification language can be extended by adding custom model transformations to the library of model transformation; addition of a model transformation should not affect the pattern instantiation engine and the proof obligation generator.

The current version of the tool is freely available from our web site [Ili08a]. Several patterns developed with this tool were applied during formal modelling of the Ambient Campus case study of the RODIN Project [IRA⁺07].

Conclusions

We proposed a theoretical basis for automation of refinement process. We introduced the notion of refinement patterns – model transformers that generically represent typical refinement steps. Refinement patterns allow us to replace a process of devising a refined model and discharging proof obligations by a process of pattern instantiation. While instantiating refinement patterns, we reuse not only models but also proofs. All together, this establishes a basis for automation. We also demonstrated how to define refinement

patterns for the Event B formalism and described a prototype tool allowing us to automate refinement steps in Event B.

Our work was inspired by several works on automation of refinement process. The Refinement Calculator tool [BGL⁺97] has been developed to support program development using the Refinement Calculus theory by R. Back and J. von Wright. [BvW98] The theory was formalised in the HOL theorem prover, while specific refinement rules were proved as HOL theorems. The HOL Window Inference library [Gru96] has been used to facilitate transformational reasoning. The library allows us to focus on and transform a particular part of a model, while guaranteeing that the transformation, if applicable, will produce a valid refinement of the entire model.

A similar framework consisting of refinement rules (called tactics) and the tool support for their application has been developed by Oliveira, Cavalcanti, and Woodcock [OCW03]. The framework (called ArcAngel) provides support for the C.Morgan's version of the Refinement Calculus. The obvious disadvantage of both these frameworks is that the refinement rules that can be applied usually describe small, localised transformations. An attempt to perform several transformations on independent parts of the model at once, would require deriving and discharging additional proof obligations about the context surrounding transformed parts, that are rather hard to generalise. However, while implementing our tool, we found the idea of using the transformational approach for model refinement very useful.

Probably the closest to our tool is the automatic refiner tool created by Siemens/Matra [BM99]. The tool automatically produces an implementable model in B0 language (a variant of implementable B) by applying the predefined rewrite rules. A large library of such rules has been created specifically to handle the specifications of train systems. The use of this proprietary tool resulted in significant growth of developer productivity. Our work aims at creating a similar tool yet publicly available and domain-independent. The idea of reuse via instantiation of generic Event B models has also been explored by Silva and Butler [SB09]. However, they focus on the instantiation of the static part of the model – the context – while our approach mainly manipulates its dynamic part. Nevertheless, these two approaches are complementary and can be integrated.

Obviously the idea to use refinement patterns to facilitate the refinement process was inspired by the famous collection of software design patterns [GHJV95]. However, in our approach the patterns are not just descriptions of the best engineering practice but rather "active" model transformers that allow a designer to refine the model by reusing and instantiating the generic prefabricated solutions.

As a future work we are planning to further explore the theoretical aspects

of the proposed language of refinement patterns as well as extend the existing collection of patterns. Obviously, this work will go hand-in-hand with the tool development. We believe that by building a sufficiently large library of patterns and providing designers with automatic tool supporting refinement process, we will facilitate better acceptance of formal methods in practice.

6.4 A Survey on Event-B Decomposition

A central idea for development using Event-B is *refinement*, where a sequence of models are constructed, with the later (concrete) one refines the previous (abstract) one. Variables v of the abstract model is linked with variables w of the concrete model by some *gluing invariant* $J(v, w)$. Consistency must be proved that any behaviour of the concrete model can always be *simulated* by some behaviour of the abstract model, with respect to the gluing invariant $J(v, w)$.

Shared-variable decomposition Consider Fig. 6.6 where machine M has events $e1$ to $e4$ and variables $v1$ to $v3$. The solid lines connect variables used by events. In Fig. 6.6(a), M is *shared variable* decomposed and events are partitioned into sub-components: $e1$ and $e2$ are allocated to machine $M1$ and $e3$ and $e4$ are allocated to machine $M2$. Consequently, $v1$ belongs to $M1$ and $v3$ belongs to $M2$ (*private variables*). Variable $v2$ is a *shared variable* between $e2$ and $e3$. Furthermore we introduce (additional) *external events* simulating how shared variables are handled in the other sub-component ($e3\_ext$ is added to $M1$ and $e2\_ext$ to $M2$). Sub-components can be refined independently but the shared variable must be present and cannot be data-refined.

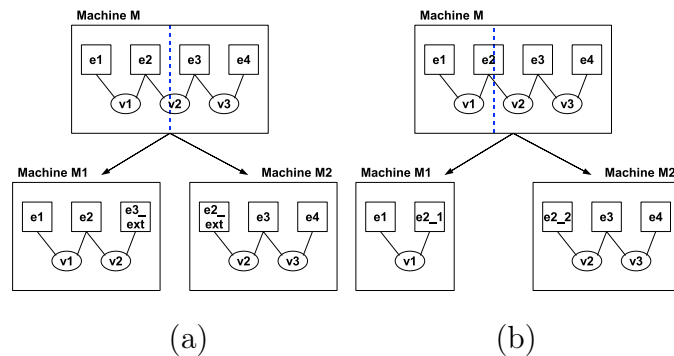


Figure 6.6: Shared variable (a) and shared event decomposition (b)

Shared-event decomposition In Fig. 6.6(b), M is *shared event* decomposed and variables are partitioned into the sub-components: $v1$ is placed in $M1$ and $v2, v3$ are placed in $M2$. Events using variables allocated to different sub-components ($e2$ shares $v1$ and $v2$) must be split. The part corresponding to each variable ($e2\_1$ and $e2\_2$) is used to create partial versions of the non-decomposed event. The sub-components can be refined independently. Shared event approach is suitable for message-passing distributed systems while shared variable approach is suitable for design of parallel algorithms [But97].

6.4.1 Guidelines

The primary challenge of a decomposition refinement step is to ensure that the structure of a model before decomposition fits the requirements of the chosen decomposition style and leads to helpful sub-models that can be developed separately with a tangible advantage in proof effort and overall model scale. As with any top-down approach for system development using refinement, the more abstract models that we have initially, the more useful our decomposition step will be.

We define guidelines for the three decomposition techniques based on the following common template.

Stage 1 Model system abstraction expressing all the relevant global system properties.

Stage 2 Refine model to fit the structure expected by a given decomposition technique.

Stage 3 Apply decomposition.

Stage 4 Proceed with independent refinement of decomposed parts.

Stage 1 and 4 are mostly the same for all three techniques. In the following sections, for each decomposition style, we give a guideline on how to accomplish the preparatory Stage 2 and the decomposition Stage 3. Shared event and shared variable decompositions place more emphasis on activities in Stage 2 whilst offering a simple and mechanised rule for Stage 3. Modularisation is not so sensitive to the structure of a model being decomposed but its decomposition stage is far more involved.

Shared-Variable Decomposition

Developing using shared-variable decomposition follows the main guideline, with the following specific details with respect to **Stage 2** and **Stage 4**. In

Stage 2, we introduce the shared elements between different components, e.g. some shared channels for distributed systems or some shared variables concurrent systems. In **Stage 4**, the sub-models can be developed independently, however, shared variables and external events can be *neither removed nor refined*.

The key important aim when designing model for shared-variable decomposition is to have the good abstract model before decomposition, i.e. at the end of **Stage 2**. Ideally, this model should contain only necessary information for modelling the global properties and the shared elements between future sub-models. Other information, e.g. control/data flows within each sub-model can/should be abstracted away. With this in mind, we can take the advantage of both refinement and decomposition as follows.

- The global properties are captured early in the model and are guaranteed to hold in the final model by the mixture of refinement and decomposition steps.
- Refinement guides the design of **Stage 2**, i.e. constraints on the shared elements will be *derived* from the need for maintaining the properties introduced in **Stage 1**.
- The development of each sub-model is done independently of the others.
- We can have different implementation for a sub-model that guaranteed to work with any implementation of other sub-models.

Shared Event Decomposition

The shared event decomposition approach allows the separation of the different states usually after proving some global properties beforehand as part of a top-down development. The global properties are preserved and the resulting sub-models can be refined by focusing only in a particular part of the system. The decomposition can be seen as an architectural decision by separating different semantics described by different state variables or as a simplification of the system when it becomes too complex to be handled as one single component. Shared event decomposition follow the general guideline steps except for the **Stage 2** where we introduce which variables (states) are allocated to each sub-model.

Like in the shared variable approach, having a good abstract model is essential as it allows to easily separate the states for each sub-model in stage 2. The further refinement of the sub-systems in stage 4 is very useful as different implementations can be developed without any additional interferences.

Modularisation

Modularisation decomposition is accomplished by demonstrating a refinement relation between an abstract model and a refinement model importing one or more modules. There are two major cases of applying modularisation based decomposition refinement:

bottom-up An interface of module is conceptually developed before the decomposition step.

top-down A module interface is naturally derived from the abstract model.

In both cases, the continuation is independent development of the decomposed parts. In modularisation, unlike SV and SE decompositions, the top-level machine is one of such parts.

Bottom-up

In this style, a decomposition step start with the formulation of one or more interfaces. A refinement chain is then planned to connect with the interfaces. In this style, some abstract variables are refined by imported module variables while the remaining imported variables become new variables of the refinement machine.

Stage 2 Construction of interface(s). There is also the stage when one draws an informal refinement plan leading to decomposition.

Stage 3 Refine the current model until there is a logical connection with the defined interfaces. Construct a refinement machine by adding the module import clause. All imported variables that are not updated via operation calls will remain read-only for all the subsequent refinement steps.

This style requires strong understanding of the system being modelling and the ability to plan refinement several steps ahead. Often such an ability comes only with a refactoring of an existing development.

Top-down

In this style a logical sub-unit of a specification is identified and decomposed into a separate module. Normally, a sub-unit is related to one or more variables manipulated in several events and it is known that the current specification of variable updates is strong enough to discharge all the top-level properties. That is, any implementation details of the sub-unit are not going to affect the provability of the later refinement steps. In this style all

the imported variables are copies or data refinement versions of a subset of abstract model variables.

Stage 2 One starts with a definition of a module interface using the current model as a guideline. The technique is to move some of model variables into a module and define operations that would replace actions updating the variables. A good strategy is try make a module interface as general (weak) as possible. This achieves two goals: a more general interface is more likely to be reused in the same or another development; if there is a mismatch between the decomposed model and the interface (as it should be expected), undischarged proof obligations give an indication of how to strengthen operation post-conditions.

Stage 3 The decomposition refinement step is a refinement importing the defined interface. Variables allocated to interfaces are declared abstract and linked with interface variable using import invariants. All actions updating on variables allocated to modules are refined into operation calls. To simplify proofs, one should try achieve a situation where the post-condition of a called operation is the same or very similar to the substitution of the refined action.

This is the main decomposition style of the modularisation approach. It makes the best use of the refinement methodology and modularisation decomposition. Action simulation proof obligation ensures that an operation of a module is related to some abstract behaviour and is used correctly. –

6.4.2 Modeling Monitor

System Description

The system contains two processes: **Poll** and **Vote**, running concurrently. They both have access to some sensors *snsrs* for updating (by **Poll**) and for reading and perform a major voting (by *Vote*). For consistency, the process can access the shared sensors only within its critical sections, established by mean of some FIFO queue. The Pseudo-code for the two processes is as follows.

```

Poll ==
c = 1
while (true) {
  // Register interest
  push(q,Poll);
  // Busy waiting
  while (head(q)!=Poll) {}
  // begin crit. sect.
  while (c /= n + 1) {
    c = c + 1
    update(snsrs(c))
  }
  pop(q)
  // end crit. sect.
  c = 1
}

Vote ==
rds, pts, st = 0, 0, F
while (true) {
  // Register interest
  push(q,Vote);
  // Busy waiting
  while (head(q)!=Vote) {}
  // begin crit. sect.
  while (rds /= n) {
    if (snsrs(rds+1) = T) {
      rds = rds + 1
      pts = pst + 1
    }
    else {rds = rds + 1}
  }
  rst = rds-pts<=pts
  st = T
  pop_queue;
  // end crit. sect.
  rds, pts, st = 0, 0, F
}

```

Shared Variable Decomposition

Before going into the actual development, we consider the some aspects that influent our design of the system for shared-variable decomposition.

- The main global property of interest is the mutual exclusive access to the shared sensors of two processes.
- We will design our system to be decomposed into *three* components: the **Poll** process, the **Vote** process and a **Mutex** component acting between the two processes for controlling their access to the corresponding critical sections. The shared elements between these components will be the sensors (which will be accessed exclusively by **Poll** and **Vote**) and the current positions of the two processes represented by their program counters.
- In the model before decomposition, we abstract away the detailed implementation of each process, e.g. how the sensors are updated by **Poll** or how the sensors are read by **Vote**. In particular, we omit the implementation of mutual exclusion using queue. The idea is that different mechanism for ensuring mutual exclusion can be deployed, e.g. Peterson's algorithm.

Stage 1. The Global Mutual Exclusion

Our global model focuses on the mutual exclusive accesses of the two processes to the sensors. Each process only accesses the shared set of sensors

snsrs within its critical section and we will deploy some mechanism to ensure that at most one process can be in its critical section at one time. Using Boolean variables p_cr and v_cr denoting if a process is within its critical section, invariant **inv0_2** captures this global mutual exclusive property. Initially, both processes are outside its critical section. Variable rst is for storing the result of the voting process later.

variables:	<i>snsrs, rst,</i> p_cr, v_cr	invariants: inv0_1 : $snsrs \in 1 \dots n \rightarrow \text{BOOL}$ inv0_2 : $p\_cr = \text{F} \vee v\_cr = \text{F}$

The sequence of events for process **Poll** is as follows. When outside its critical section, it can enter the critical section (**p_ent**). When in the critical section, it can update the value of the sensors *snsrs* non-deterministically (**p_crt**), at the moment zero or more times, before exiting the critical section (**p_ext**).

$$\begin{aligned} \text{p\_ent} &\hat{=} \text{when } p\_cr = \text{F} \wedge v\_cr = \text{F} \text{ then } p\_cr := \text{T} \text{ end} \\ \text{p\_crt} &\hat{=} \text{when } p\_cr = \text{T} \text{ then } snsrs : \in 1 \dots n \rightarrow \text{BOOL} \text{ end} \\ \text{p\_ext} &\hat{=} \text{when } p\_cr = \text{T} \text{ then } p\_cr := \text{F} \text{ end} \end{aligned}$$

The sequence of events for process **Vote** are similar, i.e. **v_ent**, **v_crt** and **v_ext**. In particular, event **v_crt** corresponding to its critical section obtains the output rst by performing a major voting (abstracted by some function mjv) on the values of sensors *snsrs*.

$$\text{v\_crt} \hat{=} \text{when } v\_cr = \text{T} \text{ then } rst := mjv(snsrs) \text{ end}$$

Stage 2. The Shared Program Counters

In this stage, we introduce the program counters for each process which will be shared amongst the future decomposed sub-models. We remove the two Boolean variables p_cr and v_cr , since now the critical sections can be defined by the value of the program counters ppc and vpc accordingly.

Each program counter can have one of the following values:

- 0** The process is waiting to enter its critical section.
- 1** The process is within its critical section.
- 2** The process is about to exit its critical section.
- 3** The process has exit its critical section.

The update of the program counter ppc is as follows.

$$\begin{aligned}
p\_ent &\hat{=} \text{when } ppc = 0 \text{ then } ppc := 1 \text{ end} \\
p\_rls &\hat{=} \text{when } ppc = 1 \text{ then } ppc := 2 \text{ end} \\
p\_ext &\hat{=} \text{when } ppc = 2 \text{ then } ppc := 3 \text{ end} \\
p\_non\_crt &\hat{=} \text{when } ppc = 3 \text{ then } ppc := 0 \text{ end}
\end{aligned}$$

To link between the abstract Boolean flag p_cr with concrete program counter ppc , the following gluing invariant is added $p\_cr = \top \Rightarrow ppc \in \{1, 2\}$. Event p_crt is refined accordingly using ppc instead of p_cr .

$$p\_crt \hat{=} \text{when } ppc = 1 \text{ then } snsrs : \in 1 \dots n \rightarrow \text{BOOL} \text{ end}$$

Guard strengthening proof obligation is trivial thanks to invariant **inv1_3**.

Stage 3. Decomposition

The summary of our decomposed models is as follows.

	Poll	Vote	Mutex
Internal events	p_crt, p_rls p_non_crt	v_crt, v_rls v_non_crt	p_ent, p_ext v_ent, v_ext
Private variables		rst	
Shared variables	$snsrs, ppc$	$snsrs, vpc$	ppc, vpc

Phase 4. Further Developments of Sub-models

The refinements of sub-models are straight forward. We give a summary of the strategy for refining for each sub-model.

- **Poll**: Introducing sensors counter c for updating the sensors one by one.
- **Vote**: Introducing variable st to denote when the result is available. Moreover variables rds and pts for calculating the result step-by-step.
- **Mutex**: Implement mutual exclusion using a FIFO queue. Alternatively, a different implementation can be used, e.g. Peterson's algorithm.

Shared Event Decomposition

The global property to be preserved is: publication of the voting results by checking the sensors (vote) and reading the individual vote sensors (poll) occur concurrently or in other words. If one of the processes is active the other is not. We design our system to be decomposed into three components: *VOTE*, *POLL* and *MUTEX* (shared area where the sensors are allocated). The implementation of the access control is deferred for after the decomposition. The decomposition focus in separating the two processes and the shared object while preserving their properties.

Stage 1. The Global Property

It is the same as the shared variable decomposition. Therefore we will just describe the following stages.

Stage 2. Introduction of Global Critical Zone

This refinement is used as a preparation step before the decomposition. We introduce a flag defining if a process is in the critical area or not (independent of which process is in the critical area), represented by the boolean variable *crt*. With this step, the processes *POLL*, *VOTE* and *MUTEX* have individual states defining the critical zone. For the shared event approach, the way each process access to the shared object has to be defined before the decomposition. Afterwards, the processes communicate with the shared object via shared parameters and there is no direct access to it. Therefore more detail is added on how *POLL* reads the sensors by introducing a pointer *c*. In order to synchronise the critical area flags, several invariants are introduced: *inv1_1* states that if both processes are outside the critical zone, then *crt* is F; in other words, if any of the processes is in the critical zone, then *crt* is T (*thm1_1*). *inv1_2* states that while the process *POLL* is not in the critical zone, the pointer is the first position.

invariants:

inv1_1 $crt = F \Leftrightarrow p\_cr = F \wedge v\_cr = F$

inv1_2 $p\_cr = F \Rightarrow c = 1$

thm1_1 $crt = T \Leftrightarrow p\_cr = T \vee v\_cr = T$

All the abstract events are refined, reflecting the introduction of the new variables. Because of *inv1_1*, in event *p_ent* the previous two abstract guards can be replaced by *crt = F* since it is implied that the none of the processes is in the critical area. Furthermore, *crt* is also updated in the actions to guarantee the consistency. A new parameter and several guards are added to event *p_crt*: now this event only reads the value of the sensor corresponding to the index *c* and is enabled until the length of the sensor is reached (*c* is constantly incremented).

<pre> p_ext refines p_ext when crt = F then p_crt := T crt := T end </pre>	<pre> p_crt refines p_crt any c where p_cr = T crt = T c ∈ 1 .. n c = c then snsr(c) ∈ BOOL c := c + 1 end </pre>
<pre> p_ext refines p_ext when p_cr = T crt = T c = n + 1 then p_cr := F p_cr := F c := 1 end </pre>	<pre> v_crt refines v_crt any s where v_cr = T crt = T s = snsr⁻¹ then rst := bool(card(s[{F}])) ≤ card(s[{T}])) end </pre>

Once all the sensor values are read, *POLL* can exit the critical zone through the event *p_ext*. Similarly, new guards are introduced stating that *crt* must be *T* and the *POLL* pointer of the sensors is equal to sensor length +1. Besides exiting from the critical zone, *c* is also reset. The events related to process *VOTE* are modified in a similar fashion to keep the consistency. In refined event *v_crt*, a parameter *s* is introduced representing the sensors (part of the preparation step before the decomposition). Therefore we are ready for the decomposition that is described in the following section.

Stage 3. Decomposition

The shared event decomposition, originates three sub-models resulting from the allocation of the variables in the original model. A semi automatic tool is provided to select which variables are allocated to which sub-model (*poll*, *vote* and *mutex*). The following table summarises the decomposition.

	poll	mutex	vote
Variables	p_cr, c	$snsrs, crt$	v_cr, rst
Events	p_ent, p_crt p_ext	p_ent, p_crt, p_ext v_ent, v_crt, p_ext	v_ent v_crt, p_ext

Stage 4. Further Development of Sub-models

The decomposition allows of separation of the processes and the access to the critical area. Therefore each process can focus only on its task and the *mutex* deals with how the access to the shared area is handled. The sub-models are refined independently as described below:

poll The refinement of poll includes the introduction of a local program counter *ppc* the possible states that the poll can be. Entering in the critical area is partitioned into smaller steps:

- *N_CRT*: process runs outside of the shared area (non-critical).
- *REQ*: process can or is requesting access to shared area.
- *ENTER*: request is acknowledged but still waiting to be granted access.
- *PROTECT*: process can or is accessing the shared area.

The state *N_CRT* is also a new part of this refinement as it refers to processing outside the shared area, represented by event *p_n_crt*. The process *POLL* works as follows: when the process needs to access the share area, it requests the permission (through a new event *p_request*) that eventually is granted (given in the refined event *p_ent*). Then the process can run in the critical area, reading all the sensors one by one until the length of *snsrs* is reached. Then this process exists from critical area using event *p_ext* and can either request new access to the shared area or enter in the non-critical area. Once it finishes running in the non-critical area (event *p_n_crt*) it can request access to the critical area again.

vote The refinement of the process *VOTE* is similar to *POLL*: a program counter *vpc* is added as well as events *v_request* to request access to the shared area and *v_n_crt* that models the access to non-shared areas.

mutex The access to the shared area is refined by implementing an algorithm that allows the exclusive access to shared data. The events `p_ent` and `v_ent` are merged into a single event `crt_ent` and similarly events `p_ext` and `v_ext` are merged into one single event `crt_ext`. This is possible because the (decomposed) merged events have the same guards and actions. From the *MUTEX* viewpoint, it is irrelevant who is requesting the access or exiting to the critical area. The exclusive access was implemented using Peterson's algorithm although other options could have been used like a queue of requests or semaphores.

Modularisation

In this case, the aim of the decomposition is to separate a protected object (sensor array) from the two processes accessing the object. The protected object provides a mechanism that prevents processes from concurrently accessing sensor values and it also encapsulate the handling of sensors value refresh. Formally, the decomposition step will reinterpret the mutual exclusion property formulated in the abstract machine but now in the terms of interface variables of the protected object module. The module interface is intentionally left abstract enough to permit differing access control strategies. One of them is based on a request queue and the other is a very simple implementation that lets processes access the protected object in some predefined order.

Stage 1. The Global Property

This stage is the initial model discussed.

Stage 3. Interface Construction

There are two major steps in this stage: definition of the interface of the protected object module and the construction of a refinement step integrating the interface. The principal idea of the protected object module is that its user may *acquire* exclusive access to the object, operate on the data of the object and then *release* exclusive access. If there is another process willing to access the object while it is locked, it will have to wait until the another process releases the object.

The interface defines two variables: the external projection of the protected data (*sn*) and the owner process (*ow*) that holds the exclusive access right (set to *none* if there is no such process).

invariants:

variables: sn, ow **inv1 :** $sn \in 1 \dots m \rightarrow \text{BOOL}$
 inv2 : $ow \in \text{OWNER}$

Initially, sn is some arbitrary vector of values and $ow = \text{none}$. A process acquires the lock by calling operation **acquire** with its name. The operation grants the access rights to the calling process if there no current owner; otherwise, it does not change the owner. To release the lock, a process calls operation **release**, also with its name. The operation changes the current owner; at the interface level, however, it is not known what process, if any, is the new owner. Finally, there is an operation requesting the update of the sensor values. It may only be called by a process owning the lock.

```

acquire
  any  proc
  pre
     $proc \in \text{OWNER} \setminus \{\text{none}\}$ 
  post
     $ow = \text{none} \Rightarrow ow' = proc$ 
     $ow \neq \text{none} \Rightarrow ow' = ow$ 
  end

release any proc pre  $ow = proc \wedge proc \neq \text{none}$  post  $ow' \in \text{OWNER}$  end

set
  any  proc
  pre
     $ow = proc$ 
     $proc \neq \text{none}$ 
  post
     $sn' \in 1 \dots \rightarrow \text{BOOL}$ 
  end

```

Stage 3. Decomposition

The decomposition step includes two activities: adding the module dependency and thus importing its variables and operations; some further detailisation of the main machine, in particular, using program counters to define process execution steps. The decomposition refinement step data refines all the abstract variables but variable rst . Abstract sensor array $snsrs$ is replaced by sn of the imported module; p_cr and v_cr are refined into more general ppc and vpc program counters of, correspondingly, processes **Poll** and

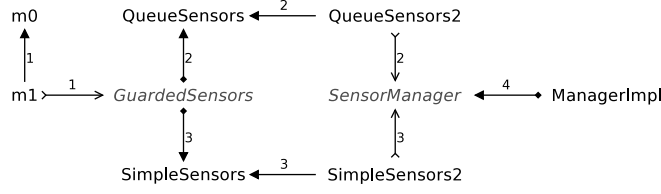


Figure 6.7: The structure of modular monitor development. Numbers 1-4 identify separate sub-developments. The arrows signify model relationships: straight arrow is refinement, fish-tail arrow is module inclusion, diamond-tale arrow is an interface implementation. *GuardedSensors* and *SensorManager* are interfaces; the rest are machines.

Vote. The following are the typing, gluing and import invariants of the decomposition refinement step.

invariants:

inv_1 : $ppc \in STAGE \wedge vpc \in STAGE$
glinv_1 : $ppc = PRO \Leftrightarrow p\_cr = \top$
glinv_2 : $vpc = PRO \Leftrightarrow v\_cr = \top$
inv_import_1 : $mon\_sn = sn$
inv_import_2 : $ppc = PRO \Rightarrow mon\_ow = Poll$
inv_import_3 : $vpc = PRO \Rightarrow mon\_ow = Vote$

Gluing invariants **glinv_1**, **glinv_2** link the value of the process program counters with the abstract notion of process critical sections. The import invariants relate the imported *mon_sn* variable with the disappearing variable *sn* and also the values of the process counters with the owner of the protected object. The latter insure that we use the protected object module correctly. Note that the mutual exclusion property proven in abstract model guarantees that only one process is the protected object owner (that is, $\neg(ppc = PRO \wedge vpc = PRO)$). We focus on the behaviour of the **Poll** process; **Vote** specification is symmetric.

Poll starts with a request to acquire the lock for the protected object; it issues the request using operation **mon_acquire** of the imported module and supplying its name as a parameter. There is little danger in mistakenly using a wrong process name due to a cross-check via process counters *ppc* and *vpc*.

$p\_req \hat{=} \text{when } ppc = REQ \text{ then mon\_acquire(Poll) } || ppc := ENT \text{ end}$

After issuing the request, a process waits until the indicated protected object owner becomes the **Poll** process. Then it may enter the critical section.

$p\_ent \hat{=} \text{when } ppc = ENT \wedge mon\_ow = Poll \text{ then } ppc := PRO \text{ end}$

For the **Poll** process, the critical section behaviour is the refresh of the sensor value. It issues a command to the protected object to fetch new sensor values. We leave out the additional details on how this event is made convergent.

$$p\_crt \triangleq \text{when } ppc = PRO \text{ then mon\_set(Poll) end}$$

Once the process is done working with the sensor values it announces the release of the lock by calling operation **mon_release**.

$$p\_ext \triangleq \text{when } ppc = PRO \text{ then mon\_release(Poll) || } ppc := REQ \text{ end}$$

The continuation depends on the specific implementation of the protected object. The protected may accumulate lock requests or have fixed process allocation policy or even simply let process to race for a lock.

Phase 4. Developments of Sub-models

This stage is concerned with providing an implementation for the protected object interface. We have construct two differing implementations, one based on requests queue and the other using a simple alteration policy (see Figure 6.7 for the overall development structure). These developments are essentially separate projects and are completely independent. We have also identified a common sub-model of these two sub-models concerned with the implementation of a sensor polling loop (Interface *SensorManager* in Figure 6.7).

Discussions

This section draws some comparison between the models developed using different decomposition techniques. Even though the high-level development strategy for the three approaches is similar, the actual details for each development are different which have subsequence effects on further development of the models in Stage 4. The essential differences between the decomposition techniques are the notion of “shared elements”. These shared elements are acting as the interfaces between the sub-components and must be respected by any future refinement of these sub-components. Subsequently, having different share elements corresponding to having different interfaces for the sub-components and this is the main differences between our approaches.

For the shared-variable decomposition, the sub-components share part of their states, e.g. the sensors and the program counters in the monitor example. Subsequence refinements of the sub-components introduce in details the related to how these shared elements are processed, e.g. how sensors are

updated or read. This refinement steps are done *independently within each sub-model*. For our example, we develop the poll process so that the sensors are updated one-by-one, refine the vote process also reads each sensor one-by-one, and implement the mutual exclusion in the mutex sub-model using Peterson’s algorithm.

For the shared-event decomposition, the sub-components synchronise on their actions and shared input/output of these actions, represented by parameters of the original events before decomposed. In particular, the state variables are partitioned into different models, e.g. the sensors belong to mutex sub-model (which will ensure the mutual exclusive access to this shared data structure). As a result, further refinement on how the sensors are updated/read and how mutual exclusion is guaranteed must be done within this mutex sub-model (in contrast with the model by shared-variable decomposition).

The development using modularisation approach contains a separate interface for accessing the shared sensors, including mechanism for ensuring the mutual exclusive access to update/read the value of sensors. In this aspect, this is similar to the share-event decomposition, however, there are no other interfaces such as poll or vote process. In principle, the main model can be refined to be more concrete. In this example, further development will essentially in the implementation of the sensors interface.

6.4.3 Modeling Update Master Data

System Description

Figure 6.8 shows a **User** process and a **Server** process keeping master data in sync. When **User** proposes a data change, it first updates its local copy, and then sends a request message to **Server** and waits for the answer. Upon receiving the request, **Server** checks the validity of the proposed change, and updates the master copy only when the change is deemed valid. Then, it sends back a response containing either an approval or a rejection. **User** has to roll back the change if a rejection is received. We are interested in the global property that the local and master databases are always identical before and after each data update procedure.

Shared Variable Decomposition

From the given diagram in Figure 6.8, we identify several aspects of the BPMN model influencing our design of the system using shared-variable decomposition.

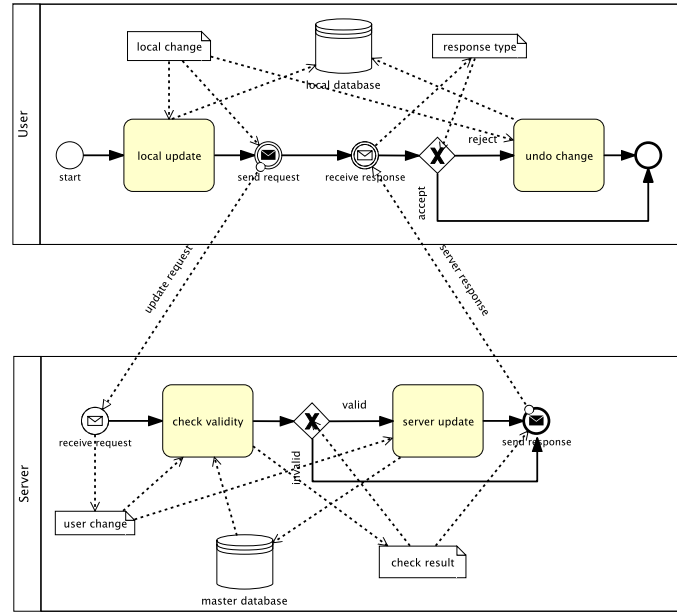


Figure 6.8: The Process Model of Update Master Data

- The main global property is to maintain the consistency between the local and server database when they are in synch.
- We will design our system to be decomposed into two components corresponding to the *local* and the *server* accordingly. The two components communicate by messages passing. The shared elements are the channels between the two sub-process.
- We will abstract as much as possible the protocol before decomposition, hiding details such as the sequencing of actions, e.g. the local updates its database *then* sends a request message; and the existence of the certain internal variables within each component.

Stage 1. The Global Property

The global property of interest is the consistency between the local and master database: when the local and master data are in synch, they must be identical. Note that it is abstract at the moment about the notion of in synch. They will be clarify later when more details of the model are introduced. In the following *DB* is a carrier set containing all the possible state of a database. In particular, invariant **inv0_3** corresponds to the previously mentioned global safety property.

variables: ldb, sdb, is
invariants:
inv0_3 $is = T \Rightarrow ldb = sdb$

Our abstract model has three events, namely **L_update**, **s_update** and **L_final**. When the system is in synch, **L_update** occurs that invalidate the in synch status. While the system is out of synch, **s_update** can happen (at this abstract level zero or more time, before **L_final** happens that put the system back in synch).

L_update	s_update	L_final
when $is = T$ then $is := F$ $ldb \in DB$ end	when $is = F$ then $sdb := ldb$ end	when $is = F$ then $is := T$ $ldb := sdb$ end

The maintenance of the invariant **inv0_3** is trivial, in particular, the action of event **L_final** ensures the system in synch by making ldb the same as sdb .

Stage 2. Shared Channels Between Components

In this stage, we introduce the channels acting in between the local and the server, and refine the variables and events into two group of events corresponding to the sites, preparing for the later decomposition step. The constraint here is that the events belonging to the local site can only reference the variables of the local and the shared channel, but not the variables of the server.

The global in synch flag, i.e. is is replaced by the lis with gluing invariant $lis = is$, i.e. the global in synch is consistent with the local view. A separate in synch flag sis is introduced for the server. Two channel for sending request and response are introduced $creq$ and $cres$. Moreover, in order to separate the local and server site completely, we introduce two new variables keeping temporary information belong to each site. For the local, variable ldb_old keeps the old value of the local database for undo later if necessary. For the server, variable sc keeps the user change to the database on the server site for updating the server database.

In the formal model below CH denotes the set of changes that can be applied to the database. The channels are modelled as a set of messages going through the corresponding channels.

variables: $ldb, ldb\_old, lis,$ $creq, cres$ sdb, sc, sis	invariants: inv1_1 : $lis = is$ inv1_2 : $ldb\_old \in DB$ inv1_3 : $creq \in \mathcal{P}(CH)$ inv1_4 : $cres \in \mathcal{P}(\text{BOOL})$ inv1_5 : $sc \in CH$ inv1_6 : $sis \in \text{BOOL}$

We keep our model abstract in the sense that information about certain control flow and data flow are omitted. For example, we assume at the moment that the update of the database and sending the request message from the local site happen simultaneously. This is represented by event `L_update_and_req`, a refinement of the abstract event `L_update`.

<code>L_update_and_req</code> refines <code>L_update</code> any ch where $lis = T$ $ch \in CH$ then $lis := F$ $ldb := upd(ldb \mapsto ch)$ $ldb\_old := ldb$ $creq := \{ch\}$ end	<code>s_receive_req</code> any ch where $sis = T$ $creq = \{ch\}$ then $sis := F$ $creq := \emptyset$ $sc := ch$ end

In event `L_update_and_req`, the local database ldb is updated to be $upd(ldb \mapsto ch)$, the new value obtain by applying changes ch ; the old local database is saved in ldb_old ; and the actual change is send as a request to the server via channel $creq$.

On receiving this request (event `s_receive_req`), the server saves the request to sc for later updating its database (if the change is valid). The server in synch flag is invalidated accordingly. This is a new event in this refinement.

The user change is either valid or invalid, depending on the case, the server either update its database and send a positive response (event `s_accept_res`), or leave its database unchanged and send a negative response (event `s_receive_req`).

<pre> s_accept_res refines s_update when sis = F then sdb := upd(sdb ↦ sc) cres := {T} sis := T end </pre>	<pre> s_reject_res when sis = F then cres := {F} sis := T end </pre>

In both case, the server in synch status is reestablished. Note that event `s_accept_res` is the refinement of the abstract event `s_update`, whereas event `s_reject_res` is a new event.

On receiving the response from the server, the local either do nothing or undo the changes made earlier by setting its local database *ldb* to be the value saved *ldb_old* earlier.

<pre> l_receive_res_acc refines l_final when cres = {T} then lis := T cres := ∅ end </pre>	<pre> l_receive_res_rej refines l_final when cres = {F} then lis := T cres := ∅ ldb := ldb_old end </pre>

Additional invariants are required in order to prove the correctness of this refinement step. Most of the invariants are *discovered* during discharging proof obligations such as guard strengthening and invariant preservation of the model. Below are some invariants related to the content of the channel and the status of the local and the server. Invariants **inv1_7** and **inv1_8** relate the server database *sdb* with the local database (current *ldb* or old *ldb_old*) depending on the content of the channel *cres*. Invariants **inv1_9** and **inv1_10** states that the local is out of synch if there are some request or response message.

```

inv1_7 :   cres = {T} ⇒ sdb = ldb
inv1_8 :   cres = {F} ⇒ sdb = ldb_old
inv1_9 :   creq ≠ ∅ ⇒ lis = F
inv1_10 :  cres ≠ ∅ ⇒ lis = F

```

Stage 3. Decomposition

This stage is semi-automatic, we provide the tool with input on how the events are partitioned into different future sub-models. Intuitively, we separate our events into two groups, corresponding to the local and server accordingly. The variables distribution amongst these model are calculated accordingly based on the information about events distribution. The summary of our decomposed models is as follows.

	local	server
Internal events	<i>l_update_and_req</i>	<i>s_receive_req</i>
	<i>l_receive_res_acc</i>	<i>s_accept_res</i>
	<i>l_receive_res_rej</i>	<i>s_reject_res</i>
Private variables	<i>ldb, ldb_old, lis</i>	<i>sdb, sc, sis</i>
Shared variables	<i>creq, cres</i>	<i>creq, cres</i>

Stage 4. Developments of Sub-models

In this stage, the resulting sub-models from the previous decomposition steps can be developed in dependently. We present a summary of the additional refinement steps for each model. Most invariants in the sub-models are technical and related to the sequentialisation of the actions, reflecting the BPMN diagrams.

local The control flow is introduced via means of a program counter *lpc*. In particular, updating database and sending request is done in two steps. Similarly, receiving a response and undoing the change or leaving the database unchanged are also splitted. In order to accommodate the splitting according to the control flow, the abstract database *ldb* is replaced by the actual local database *ldb_actual* and requires that when the local is in synch, their value the same (think of *ldb* as an abstract view of the actual database *ldb_actual*). This is a standard Event-B refinement technique for introducing sequential steps. Other internal variables of the local are introduced accordingly to the BPMN model, i.e. the local change *lch* and the local response type *lres*.

server Similarly, the control flow of the server is introduced via means of a program counter *spc*. The actual database *sdb_actual* is introduced, together with internal variables of the server, such as the check result *scr*.

Shared Event Decomposition

From the given diagram in Figure 6.8, we identify several aspects of the system influence our design of the system using SE decomposition. The global property is the same as before: consistency between the local and server database when they are in synch. We also design our system to be decomposed into two components: local and server. The two components communicate by message passing. The content of the message is the data that needs to be changed and the server's response about the validity of the requested modification.

Stage 1. The Global Property

The global property of interest is the consistency between the local and master database: when the local and master data are in synch, they must be identical. The variables, invariants and the event *init* are similar to the shared variable decomposition. Our abstract model has three events, namely **L_update**, **s_update** and **L_final**. When the system is in synch, an occurrence of **L_update** invalidates the in synch status because a modification (represented by parameter *d*) occurs. While the system is out of synch, **s_update** can happen (zero or more times), before **L_final** happens reversing the system back to synch.

L_update any <i>d</i> where <i>d</i> ∈ <i>DB</i> <i>is</i> = T then <i>is</i> := F end	s_update any <i>d</i> where <i>d</i> ∈ <i>DB</i> <i>is</i> = F then <i>sdb</i> := <i>d</i> end
L_final when <i>is</i> = F then <i>is</i> := T <i>ldb</i> := <i>sdb</i> end	

The maintenance of the invariant **inv0_3** is trivial, in particular, the actions of event **L_final** ensures the system is in synch by making *ldb* the same as *sdb*.

Stage 2. Introduction of the Message Passing Protocol

We introduce the channels and respective message passing protocol between *ldb* and *sdb*. Variables and events are refined into separated parts as a preparation step before the decomposition. The goal is to have variables related to *ldb* in one side and variables related to *sdb* in the other.

The global flag *is* is replaced by two individual flags referring to each side: *lis* for the local synch and *sis* for the server sync. States are introduced in the server:

- *S_IDLE*: server is idle, waiting for a request
- *RCV_RQ*: server received a request
- *VAL_RQ*: validate a request
- *S_PRC*: server is processing

A request also has states:

- *IDLE*: not request, so idle
- *PRC*: request is being processed
- *VALID*: valid request
- *INVALID*: invalid request

The protocol follows as described: *ldb* is initially idle. When a modification happens locally, it is stored locally and a modification request (message passing) is sent to the server. If the modification is considered valid, then *ldb* commits the modification; if it is invalid, the modification is discarded. Note that here we save the modification and only when the response is received, *ldb* may be modified. From the server side that initially is idle, when a request is received, it must be validated. If the request is considered valid, the modification is reflected in the server. If not, the modification is ignored. In either case, a reply is sent back with the validation result.

The synchronisation depends on the value of the variables *lis* and *sis*: the gluing invariant between *lis*, *sis* and *is* is given by invariants *inv1_1*, *inv1_2* and *inv1_3*: *lis* always matches *is*; while the modification request is being processed, *sis* matches with *is*; when not processing the request, the server is synchronised (*sis* = T).

variables: $ldb, sdb, lis,$ $l\_chg, rqst\_res$ sis, s_stt, s_chg	invariants: inv1_1 $lis = is$ inv1_2 $rqst\_res = PRC$ $\Rightarrow sis = is$ inv1_3 $rqst\_res \neq PRC$ $\Rightarrow sis = T$

init

begin

$ldb, sdb :| ldb' = sdb'$

$lis, sis := T$

$s\_stt := S\_IDLE$

$l\_chg, s\_chg : \in CH$

$rqst\_res := IDLE$

end

Some control variables are added to keep the separation between the parts: l_chg (element of set CH) corresponds to the modification in ldb ; $rqst\_res$ holds the state of the modification request from a local point of view; s_stt corresponds to the server state and s_chg holds the change (element of set CH) to be modified on the server side. In this refinement, the occurrence of a modification is modelled by the refined event *updateLocalDB*. The data to be changed is stored in l_chg and after sent to the server by the new event *send_rqst*. This event simultaneously sends the message locally and receives in the server, where it is stored s_chg so the server can validate the request. Once the server receives the request, the server is out of sync. The update is expressed by a function *upd* defined as: $upd \in (DB \times CH) \rightarrow DB$. It receives as arguments a value of an element of DB and an element of CH and returns the new DB updated with the change.

l_update refines l_update any c where $c \in CH$ $lis = T$ $rqst\_res = IDLE$ with $d = upd(ldb \mapsto c)$ then $lis := F$ $l\_chg := c$ end	send_rqst any msg where $lis = F$ $msg = l\_chg$ $rqst\_res = IDLE$ $s\_stt = S\_IDLE$ then $s\_chg := msg$ $s\_stt := VAL\_RQ$ $sis := F$ $rqst\_res := PRC$ end

We abstract from the validation of the request at the moment deferring to after the decomposition. If the request is considered valid, then the server is updated. This is modelled by the refined event `s_update`. Note that the abstract parameter `d` is replaced by the new `DB` after applying the modification `s_chg` and expressed as $upd(sdb \mapsto s\_chg)$. Even when the request is deemed *INVALID*, a response is sent back which is modelled by the new event `rcv_rsp`. This event syncs in the server that goes to mode idle and updates the request validation in the local side through the parameter `rst` (message passing). A valid request implies that the server processed the modification corresponding to the state *S_PRC*.

<pre> s_update refines s_update when sis = F s_stt = VAL_RQ with d = upd(sdb ↦ s_chg) then sdb := upd(sdb ↦ s_chg) s_stt := S_PRC end </pre>	<pre> rcv_rsp any rst where rst ∈ {VALID, INVALID} rst = VALID ⇔ s_stt = S_PRC s_stt ∈ {S_PRC, VAL_RQ} rqst_res = PRC sis = F then rqst_res := rst s_stt := S_IDLE sis := T end </pre>

On the local side, the result of the request defines what should be done: if the request is valid, the modification `l_chg` is applied locally; if the request is invalid, the `l_chg` is discarded and `ldb` remains the same. In either case, the `ldb` is back in sync with the server and the result of the request is reset ($rqst\_res := IDLE$). This is modelled by the refined event `l_final`.

```

Lfinal
  refines Lfinal
  any u where
    lis = F
    rqst_res ∈ {VALID, INVALID}
    rqst_res = VALID ⇒ u = upd(ldb ↦ l_chg)
    rqst_res = INVALID ⇒ u = ldb
  then
    ldb := u
    rqst_res := IDLE
    lis := T
  end

```

New invariants are introduced to verify the new variables and events are a valid refinement. The gluing invariants are discovered as a result of the generated proof obligations. For instance, *inv1_8* states that while the *lis* is false and a *rqst_res* is being processed, the modification data is the same locally and in the server; *inv1_9* states that if the request is considered invalid, *ldb* matched with *sdb*; if the request is valid, then the modification applied locally matches with the server as stated by *inv1_10*; *ldb* and *sdb* match as long as the request is not valid and it is not being processed by the server as stated by *inv1_11*; local and server states are related by *inv1_12* where it is stated that if a request is being processed, that means that the server is validating the request or processing the modification and vice-versa; if the server is in the processing state, server has been updated similarly to update *ldb* with the change *s_chg* described by *inv1_13*. *inv1_14* states that if the request is being processed but the server is yet not processing the modification, the resulting *DB* from modification of *ldb* by *l_chg* and *sdb* by *s_chg* are the same.

```

inv1_8 : lis = F ∧ rqst_res = PRC ⇒ s_chg = l_chg
inv1_9 : rqst_res = INVALID ⇒ ldb = sdb
inv1_10 : rqst_res = VALID ⇒ upd(ldb ↦ l_chg) = sdb
inv1_11 : rqst_res ≠ VALID ∧ s_stt ≠ S_PRC ⇒ ldb = sdb
inv1_12 : s_stt ∈ {VAL_RQ, S_PRC} ⇔ rqst_res = PRC
inv1_13 : s_stt = S_PRC ⇒ sdb = upd(ldb ↦ s_chg)
inv1_14 : rqst_res = PRC ∧ s_stt ≠ S_PRC
          ⇒ upd(sdb ↦ s_chg) = upd(ldb ↦ l_chg)

```

This model is synchronous but could easily be modelled in an asynchronous manner by introducing a buffer in between the local and the server with two different channels: one to send the modification to the server and the other

to send the validation result back to the local side. That would suggest a three way decomposition in opposition to the two way that we propose here.

Stage 3. Decomposition

In a shared event decomposition, the sub-models result from the allocation of the variables in the original model. A semi automatic tool is provided to select which variables are allocated to which sub-model (*local* and *server*). The following table summarises the decomposition.

	local	server
Variables	<i>ldb, l_chg, lis, rqst_res</i>	<i>sdb, s_stt, sis, s_chg</i>
Events	<i>l_update, l_final</i> <i>send_rqst, rcv_rsp</i>	<i>s_update</i> <i>send_rqst, rcv_rsp</i>

Stage 4. Developments of Sub-models

The decomposition allows the separation of sending the modification and receiving in the request in the server by splitting the event *send_rqst*. Similarly the response is separated after the decomposition of the event *rcv_rsp*. Nevertheless the sub-models can be further refined independently:

local In the local side, a program counter is added defining the local states (update *ldb*, send modification request, receive modification response, commit/discard modification). Furthermore event *l_final* is refined by two concrete events: *l_commit* when the request is deemed valid and the modification updates *ldb* and *l_discard* when the request is invalid and the modification is discarded. Another refinement that can be implemented is to add a queue of requests and in that case, there is no need to wait for the reply of the server before the following modification.

server The server is refined by modelling the request validation. Variable *val* of type *VAL_STT* and new event *s_val* are added. Similar to the local refinement, the server also can receive multiple requests and introduce them in a queue before the validation and processing.

Modularisation

The modularisation version takes quite a different approach than the other decomposition styles. We start with a top abstract machine that does not

focus on the global property. Instead, it describes abstract message flows. Interfaces are then defined for the two processes, which deliver certain promises in form of the operation post-conditions. They are implemented by separate machines that detail local control and data flow information of each process. These implementation machines are hidden from the global machines. A sufficient amount of implementation details, such as local variables and properties, is carefully chosen and exposed in their interfaces to enable the verification of the global property. Last, the top abstract machine is refined by adding details of operation calls and message buffers. The property is then verified on the final refinement machine.

Stage 1. Specifying Abstract Message Flows

The top abstract machine specifies abstract message flows, using flags to indicate whether a certain message has been sent or received. All flags are initialized to false, and set to true by corresponding message events. We also use invariants to describe the order of message events. These invariants are needed later for property verification. Note that we give no detail of how messages are actually sent, stored, and received.

```

variables:
    request_sent, request_received,
    response_sent, response_received

invariants:
    inv5  request_sent = F
            $\Rightarrow$  request_received = F
    inv6  request_sent = F
            $\Rightarrow$  response_sent = F

    send_response
    when
        request_received = T
        response_sent = F
    then
        response_sent = T
    end

```

Stage 2. Process Interfaces

For each process, its interface defines a set of message operations. These operations do not perform actual message sending or receiving. They just

provide messages to be sent (**get_request**), and take incoming messages fed to them for local consumption (**put_response**).

```

variables:
    initDB, localChange, serverResponse, finalDB,
    request_sent, response_received

    get_request
        pre
            request_sent = F
            response_received = F
        return
            msg
        post
            msg' = localChange
            request_sent' = T
        end

    put_response
        any msg
        pre
            msg ∈ BOOL
            request_sent = T
            response_received = F
        post
            serverResponse' = msg
            response_received' = T
        end

    get_final_db
        pre
            request_sent = T
            response_received = T
        post
            finalDB' ∈ N
            serverResponse = T ⇒ finalDB' = localChange
            serverResponse = F ⇒ finalDB' = initDB
        end

```

The operation **get_request** produces a message equivalent to the local change proposed by the user, as stated in its post-conditions. This provides a crucial link between the local variables of two processes. The operation **get_final_db** can only be called after the two message operations are already

executed. It gives the final value in the local database based on the type of the response received from the server. Due to space limitation, the server interface is not shown here, which is very similar to the user interface.

Stage 3. Process Implementations

The implementation of each process describes the local control and data flow information. Activities and gateways inside a process are modeled as events that change control states and update local variables. Message-related flags are no longer present, thus we provide a link between those flags and local control states as gluing invariants. The modularisation plug-in generates proof obligations to ensure that all promises in the interface are delivered by the implementation. The process implementations here are very similar to those in other decomposition approaches, so we omit the details here.

```

variables:  user_cs

invariants:
  inv18  request_sent = F
           $\Leftrightarrow user\_cs \in \{user\_cs\_start, user\_cs\_upd,
          user\_cs\_req\}$ 
  inv19  response_received = T
           $\Leftrightarrow user\_cs \in \{user\_cs\_gate, user\_cs\_undo,
          user\_cs\_end\}$ 

```

Stage 4. Final Global Machine

The top abstract machine is refined to contain operation calls and actual message exchanging behavior. Each process has a buffer to store incoming messages. When a message needs to be sent, it calls the corresponding interface operation to retrieve the outgoing message and adds it to the corresponding buffer. When a message is to be received, the message is taken out of the buffer and passed to the local process through the respective interface operation.

```

send_response
  when
    server_request_received = T
    server_response_sent = F
  then
    buf_user = buf_user  $\cup$  {server_get_response}
  end

```

Deadlock and Livelock Freedom

Deadlock freedom is proved at two levels: (a) At the final global machine, we prove that, before data sync ends, some events are always enabled at any time. This is proved through ensuring that expected messages always arrive. (b) At each process implementation machine, we prove that, when an interface operation is called, at least one of the events in the corresponding group is always enabled (see one example invariant below).

invariants:

inv20 $request\_sent = F$
 $\wedge response\_received = F$
 $\Rightarrow user\_cs \in \{user\_cs\_start, user\_cs\_upd,$
 $user\_cs\_req\}$

Livelock freedom is also proved at two levels: (a) At the top abstract machine, we define the following variant that decreases every time an event is executed. (b) At each process implementation machine, we prove that each event group is convergent toward the final event of the corresponding operation. This is done by defining a total order of the control states.

variant:

$card(T, request\_sent)$
 $+ card(T, request\_received)$
 $+ card(T, response\_sent)$
 $+ card(T, response\_received)$

Property Verification

The proof of the global property consists of two steps. (a) The interfaces of the two processes provide symbolic expressions of the final values of the local and server databases. We need to prove that these symbolic values are actually delivered by the implementation machines. This is done by adding auxiliary invariants in a symbolic execution fashion to obtain the symbolic runtime values of local variables at each control state. (b) At the final global machine, we prove that the symbolic values of the local and server databases are equivalent. This is done by building links between local variables in the two process interfaces through message flows.

Discussions

All approaches decompose global machines into two components, one for each process. Shared-variable and shared-event decompositions start with a common abstract machine, specifying a minimal set of events reflecting how the local and remote databases are updated in a way to satisfy the global property of interest. Once proved, this property is preserved through a series of refinements with appropriately chosen gluing invariants and the proofs of deadlock freedom and convergence. These two approaches are very similar, apart from that shared-event decomposition decides to model synchronous message passing. The choice is mostly motivated by the fact that, in our particular example system, the two processes are totally independent of each other, and the only links between them are achieved by message passing. Thus, it is a natural choice to have message synchronization that can be shared by both processes. However, as pointed out previously, it is also possible to model asynchronous communication for our system using shared-event decomposition. But then we would need to design the message passing behavior as a separate component that interacts (and synchronizes) with other processes.

While the other two approaches are pure top-down designs, the modularisation model blurs the boundary and can be taken (or further developed and maintained) as in a bottom-up fashion. This is due to the fact that the only connections between global machines and local components are interfaces. One advantage that immediately comes to mind is flexibility. For example, when new details or even new components need to be added to the model (especially to the global machines before decomposition), it is possible that existing components (together with their proofs) will not be affected. The largest challenge of modularisation is the design of appropriate interfaces as they play a crucial role in linking multiple worlds while proving the global property. During the design of our model, we go through iterations to “trial and error” to find the appropriate amount of information that need to be exposed in the interfaces.

6.4.4 Conclusion

Related Work As mentioned in [HA10], shared-variable decomposition is similar to rely/guarantee approach from Jones [Jon83b]. A pair of internal/external events is essentially an encoding of a rely/guarantee conditions. Technique for decomposition also appeared in the work of Abadi/Lamport [AL95]. Whereas Lamport later claimed that decomposition is not be that useful [Lam97], we believe that the difficulty can be overcome by our guidelines for develop-

ers to come up with useful models for decomposition. Compared to Action System [Bac90], shared-variable decomposition corresponds to *concurrent action systems*, whereas their notion of *distributed action systems* is similar to shared-event decomposition [BS89]. Hence our guidelines should be applicable for developing systems using Action System. Several works try to exploit the decomposition benefits: [BB06] study the formal development of MAS (Multi-Agent Systems) which are complex distributed systems to be used for critical applications using abstraction and decomposition for classical B and Event-B. Butler [But02] uses the shared event approach in classical B to decompose a railway system. The system is modelled and reasoned as a whole in an event-based approach, both the physical system and the desired control behaviour. Rezazadeh and Butler [RB03] use classical B to model a distributed monitoring and control system for vehicles entering and leaving a controlled area. After some refinements, the model is decomposed into asynchronous subsystems. That formal modelling considers only safety properties and a decomposition is suggested based on the CSP style message-passing channels. The shared event decomposition uses synchronisation and communication in the CSP style. Moreover CSP value passing channels correspond to sub-components events that communicate via shared parameters.

6.5 Formal Feature-Oriented Reuse in Event-B

6.5.1 Introduction

A software product line (SPL) is a set of related products which share a common base having significant variabilities to meet user requirements [CN01]. Each member of an SPL differentiates it from other members in terms of functionality or behaviour but share the common base. SPL development promises benefits of reusability, i.e., improved quality and reduced effort and time to market when building similar products by reusing parts of the systems that have already been developed. Feature modelling [CAK⁺05] has been established as a technique for modelling commonality and variability in SPLs. A feature has been defined to be a functionality or behaviour that is of some value to a stakeholder [CHS08]. It can be a module in a programming language or a group of requirements in the requirements specification document. A feature model is a hierarchically arranged set of product line features [Bat05]. It consists of a tree structured feature diagram with various features of a product line (PL) as tree nodes.

We have developed a framework that combines the strengths of both formal methods and SPL engineering, inspired by the work of Snook et al.

[SPJ08]. This means that if we build a database of Event-B models for a PL that have already been proved, we can specify various products of the PL by configuring and composing these models in different ways. We propose guidelines and tool support through which the process of configuration and composition of Event-B models (with various refinements) would save lot of user effort, time and reduce overall development cost.

Our contribution presented in this section is in providing methodology for feature-based reuse in Event-B, exploiting its existing capabilities, building further tools and techniques and suggesting modelling guidelines for SPL development using Event-B. We have provided notations for feature-modelling in Event-B to specify a feature model of a product line with variabilities. We considered an Event-B development (collection of models) as a ‘feature’. Other possibilities of finer granularities of a ‘feature’ in Event-B were reported in our earlier work [PFF⁺08]. In feature modelling, features are atomic whereas we proposed composition and instantiation mechanism for features which are Event-B developments. After drawing a feature model, it could then be configured to model a particular instance of a product line. The feature model could be expressed with various constraints and how the features could be composed in a valid way to produce a PL member. We have developed a feature modelling tool (inspired by earlier feature modelling tools, e.g., [AC04]) as a plug-in to Rodin that can be used to draw feature models and configure these feature models that includes conflict resolution and composition of selected features. Our case-study examples showed that we can utilise existing Event-B (de)composition techniques to decompose a system into various features, refine them and recompose later to model the desired PL instance. The two case-studies modelled different systems in different styles to explore reusability and at the same time Event-B’s potential for feature-modelling. These also highlighted further tooling requirements and research questions. We have also proposed some guidelines that we think would be useful for SPL modelling using Event-B’s current tools and techniques.

6.5.2 Feature Modelling

Introduction

Feature modelling is commonly used these days for SPL engineering. It was proposed as part of the feature-oriented domain analysis (FODA) method [KCH⁺90] in the early 90’s. “Feature modelling is a technique for representing the commonalities and the variabilities among a set of systems in concise, taxonomic form” [CAK⁺05]. It can be considered as the activity of identifying

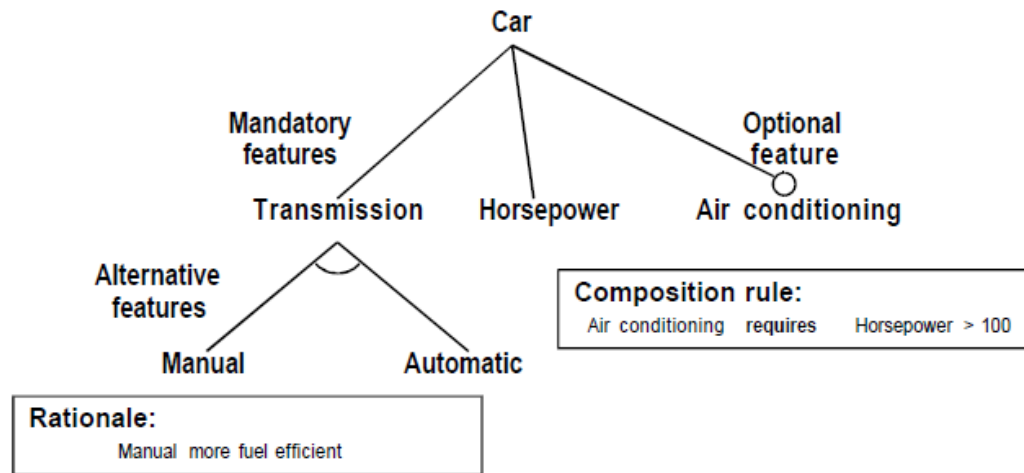


Figure 6.9: Example of a Feature Model (FODA style) [KCH⁺90]

externally visible characteristics of products in a domain [LKL02]. It can also be used for scoping and developing domain-specific languages [CHE05].

Feature modelling provides means to organise features into a feature model and configure them in order to build products of a product line. One of the many definitions of a *feature* is: “a logical unit of behaviour specified by a set of functional and non-functional requirements” [Bos00] and usually referred as a property of the system that is of some value to the stakeholders. It is considered as a unit of reuse, specialisation and composition (usually a piece of code in a programming language, e.g., a module) in SPL engineering. A feature has also been defined as an increment in program functionality [Bat05].

Existing Feature Modelling Notations & Tool Support

There are various notations used for feature modelling which extend the original feature modelling notations from FODA. A feature model consists of one or more feature diagrams. The most basic of these notations include a feature diagram which represents a product family. A feature diagram is a hierarchical tree structure of all possible features that may occur in a product. The PL is the set of all distinct products that can be instantiated from the feature model. A feature can be optional, mandatory or mutually exclusive to another feature. Various composition rules are also supplied in textual form along with the feature diagrams. Figure 6.9 shows an example of a basic feature model drawn using FODA notations.

One of the common notations for feature modelling currently used is

‘cardinality-based feature modelling’ by Czarnecki et al. [CHE05]. They extended the FODA notations by introducing feature and group cardinalities, feature attributes and feature diagram references. A feature diagram has a root feature containing further features. The cardinality of a feature or a group is provided as an interval (e.g., $m..n$). Features can refer to a feature diagram which improves reusability by referring to the feature diagram at various points in the feature model. This feature diagram reference is similar to our feature inclusion constraint discussed later in our approach.

Different configurations of features in a feature model will result in different instances of the system or members of the product line. This is done through the feature configuration diagram where the user selects the desired features to be included in the feature model instance.

Several tools have been developed to support feature modelling for product line software engineering [AC04, LKL02]. Some of these focus on the demonstration of their extended notations and others implement the existing notations in different perspective and for different domains. Some of these also reflect on the improvements from the previously existing tools.

6.5.3 A Feature Modelling Approach for Event-B

We adapted and extended the ‘cardinality-based feature modelling’ notation [CHE05], so that we can build an Event-B specific feature modelling tool as a Rodin plug-in. The reason for doing so is because it was difficult to use existing feature modelling tools to specify Event-B features. A tool specific to Event-B would be able to adopt any modifications to the language as it is continually being improved. Also, this would enable us to make use of all the available plug-ins of the Rodin platform such as editors, provers, model checkers and animators. Our recent experiment of adding Event-B support to an existing feature modelling tool (FeatureHouse [AKL09]) did not provide the same flexibility of feature configuration and composition as with our proposed feature modelling tool. It has proved difficult to integrate independently developed tools within the Rodin. One reason was an incompatibility of model interchange format between FeatureHouse (plain text) and Rodin (Event-B syntax aware XML). We would also require refactoring support while composing features and prefer to use a graphical feature composition tool rather than the script-based composition approach of FeatureHouse. So, our proposed feature modelling approach will be fully supported by a state of the art formal method.

Our Feature Modelling Notations

We define a *feature* to be an Event-B development which consists of a machine with various refinements and their seen context(s). This definition of feature is different to the one presented in our earlier work [GPBS10], where we considered a single machine and its context as a feature. Since an Event-B feature can now have a chain of refinements, that makes it more difficult to deal with composing features having various refinement levels. We will discuss this later in our case-study examples.

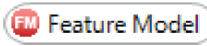
	Mandatory		Group with cardinality
	Optional		Includes
	OR Group		Excludes
	XOR Group		

Figure 6.10: Our Feature Modelling Graphical Notations

The graphical notations used in our feature modelling framework are given in Figure 6.10. A feature model consists of a tree structured feature diagram whose root feature takes the name of the model. The filled circle on a feature shows that it is a mandatory feature and optional otherwise (empty circle). The features with a triangle attached represent group features which are containers for other features and specify any constraints on the selection of features within that group. One such is the *cardinality* constraint that indicates how many of the features in the group must be present in a particular instance. An empty triangle means exclusive OR (XOR) or otherwise a group with cardinality (e.g., ‘2..4’) and the filled triangle means OR, i.e., the cardinality ‘1..k’, where k is the number of features in that group. There are three types of connections that can be used to connect various model elements: child features, includes and excludes. The *includes* and *excludes* serve as constraints in the feature model. A feature can include other features, i.e., selecting that feature must also select the included features. Similarly, a feature can exclude other features and it is mutually exclusive, which means you can not have any two features with excludes connection between them in the configured instance. The leaf level features are actually mapped to Event-B developments during configuration.

Figure 6.11 shows an example feature model drawn using our feature modelling notation. The root feature PC has a group of five features with cardinality “5..5” which means an instance must select all five of the group features. The features *BeltCrane* and *AdvCrane* of the *CraneType* group are

mutually exclusive (i.e., cardinality ‘1..1’ shown by an empty triangle) which means both of these can not be present in a particular variant of PC derived from this feature model. By selecting *BeltCrane* feature, both the *DepositBelt* and *Crane* become mandatory due to group cardinality constraint, i.e., ‘2..2’.

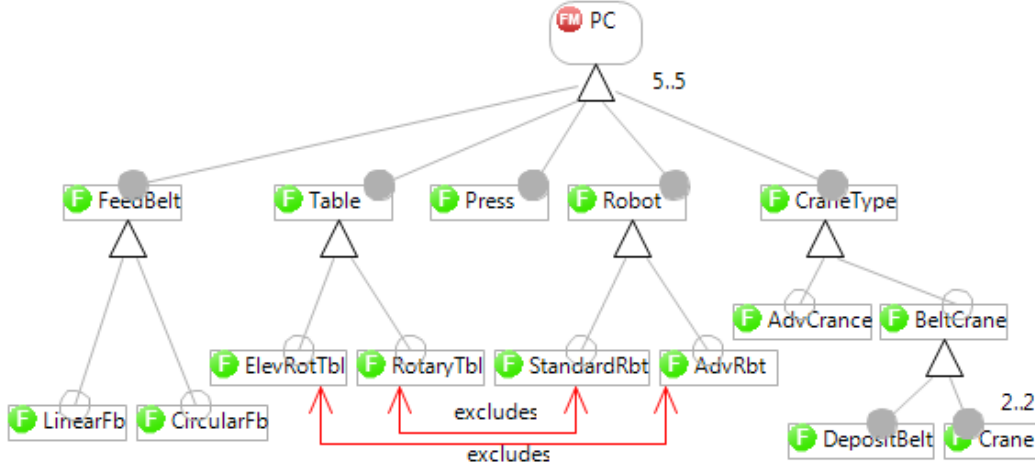


Figure 6.11: Example Feature Model

Our main extension to existing notations is that of including the refinement concept of Event-B. We consider an Event-B development as a feature, where each refinement model can be composed during the configuration for product line instantiation. We also provide *includes* and *excludes* constraints. Our feature modelling notation slightly differs from [SPB10], as we allow feature refinements and a feature can exclude other features in a feature model. On the other hand, that work considers the reuse of proof obligations for invariant preservation and feasibility; we have not yet examined this.

6.5.4 Case-Study Experiments

The case-study experiments were carried out to explore whether existing Event-B tools and techniques can be used for feature-oriented modelling in Event-B, to gain the benefits of reusability. We would see how the Event-B methodology of single system, top-down refinement based development with decomposition can be combined with multi-system, side-ways in, compositional, reuse-oriented approach of feature modelling for SPL engineering. The case-studies would also show the opportunities and constraints for feature-based working in Event-B and would help in suggesting tools and techniques for future.

We have modelled two case-studies in Event-B, i.e., production cell (PC) and ATM. Our first case-study (PC) suits the typical top-down refinement approach of Event-B. At first sight, this case study does not show obvious reuse potential in terms of functional requirements features; variability arises as different possible connection topologies between PC components. We then model it in a different way by considering a fine-grained view of features as controllers that were generic and revealed more reuse opportunity for PC modelling. We would explain how we modelled PC in different ways to explore Event-B's capability for feature modelling using its decomposition and genericity techniques to exploit reuse. In order to support our findings of the PC case-study, we then modelled the second case-study(ATM). ATM example is more reuse-oriented in term of functional features and suits the traditional feature modelling approach where we can have a product line of ATM systems having different features. This has revealed a pattern of modelling that could be used as a guideline for SPL modelling in Event-B for future users. Both case-studies will use existing (de)composition approaches of Event-B discussed earlier. In this chapter we present only the first case study – production cell (PC). A detailed description of the ATM case study can be found in [GPB11].

We have developed a prototype feature composition tool [GPS09] which was used for composing Event-B feature during the case-study experiments. This tool allows the free-style composition (including fusion) in whatever ways the user wants but it does not automatically discharge proof obligations for the composite model. This means the user need to reprove the composite model to make sure that the composition was performed correctly. The case-study results will show how we can avoid reproof effort by following a particular modelling pattern involving different (de)composition styles.

Production Cell

The production cell (*PC*) [Lin95] is an example of a reactive system which has been modelled in more than 30 formalisms [LL94, OL07, PCB98, Die96]. It has also been specified in the B formal method which is a predecessor of the Event-B language [Sek98]. It is a metal processing production line where metal blanks are routed to a press for forging, then routed away from it after processing. Figure 6.12 shows the top view of the production cell plant. Metal blanks enter into the system through the feed belt and are dropped on to the elevating-rotary table when the table is empty and in the loading position. The table elevates and rotates to a position so that the first robot arm can pick up the blanks as the robot arms are positioned at a different horizontal plane. The robot rotates anti-clockwise to drop the blanks in the

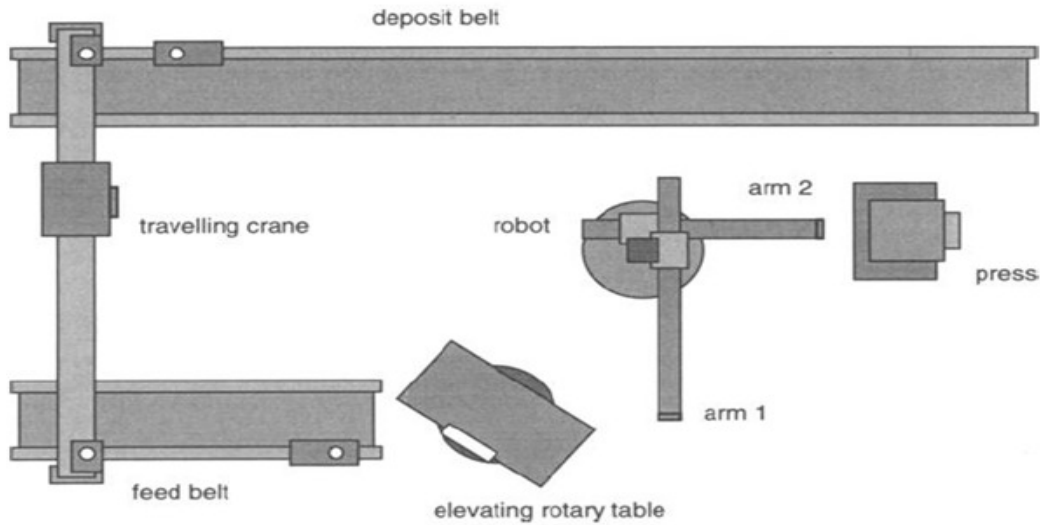


Figure 6.12: Production Cell Plant [Lin95]

press. The press forges the blanks which are picked up by the second robot arm and then dropped on to the deposit belt. A moving crane then picks the blanks from the deposit belt, that have not been forged properly, and brings them back to the feed belt for reprocessing.

We have modelled PC in three ways, i.e., physical component-based, controller-based (functional) and a domain-specific modelling approach based on static variability as described below in detail. This allows us to use different methods of modelling the same system in Event-B and analysing our approaches to feature-based modelling using existing tools and techniques in Event-B.

PC Component-based: In the physical component-based modelling approach, we started with an abstract model of the production cell and refined up to a few levels by adding more details in each refinement step. At the abstract level (see Figure 6.13), we only have one event *Operate* which models the processing of blanks from ‘forged’ to ‘unforged’ state through the variable ‘blanks’. Initially, all the blanks are set to ‘unforged’. We refined this model by introducing various physical components of PC (e.g., table, robot, press etc.) and where the blanks are positioned across these components. This is further refined by modelling operations taking place at different components of PC, e.g., the elevation and rotation of table etc. In the next refinement, we introduced control functionality of robot arms and the movement of belts for delivering blanks. Some safety requirements were also modelled such as arms should not be extended while the robot is not positioned correctly. Another refinement was done to introduce the

```

machine PC_0 sees PC_C0
variables blanks
invariants
  @invt blanks ∈ BLANKS → STATUS
events
  event INITIALISATION
  then
    @act1 blanks = unfBlanks
  end
  event Operate
  any b
  where
    @grd1 b ∈ dom(blanks)
    @grd2 blanks(b) ≠ forged
  then
    @act1 blanks(b) = forged
  end
end
end

```

Figure 6.13: PC Abstract Machine

functionality of the ‘crane’ component. These were all horizontal refinement steps.

At this stage, the model became quite large and was difficult to refine further as a whole. So, we decomposed the model into various physical components (sub-models) of the PC (i.e., *feed belt*, *table*, *robot*, *press*, *deposit belt* and *crane*). We used both types of decomposition, i.e., shared-variable (*SVD*) and shared-event decomposition (*SED*). At first, it seemed appropriate to use SVD since different components were sharing variables, e.g., all components shared the variable *blanks*, which models the status of blanks at any component. So, during the decomposition, events related to a particular physical component became events of that sub-model and any events of the sub-model involving the shared-variable became external events in all the other components. For example, event *loadTable*, moved to the *table* sub-model, became an external event in all the sub-models for other physical components of PC and so on.

In order to explore whether we can use SED to decompose the integral model into sub-models, we had to prepare the model to be decomposed using the SED style. For this, we had to partition(localize) the variable *blanks* for each component (i.e., *blanksOnFb*, *blanksOnTbl* etc.), so that there is no more shared-variable. In this case, we partitioned the variables into various sub-models along with their related events. Figure 6.14 shows how an event *loadTable* is split into two events, i.e., *loadTableF* and *loadTableT* for *table* and *feed belt* components. We simply split the guards and actions into two. If a guard or action of an event is complex and can not be split then it must be simplified in the preparatory step to be split into two. Note that

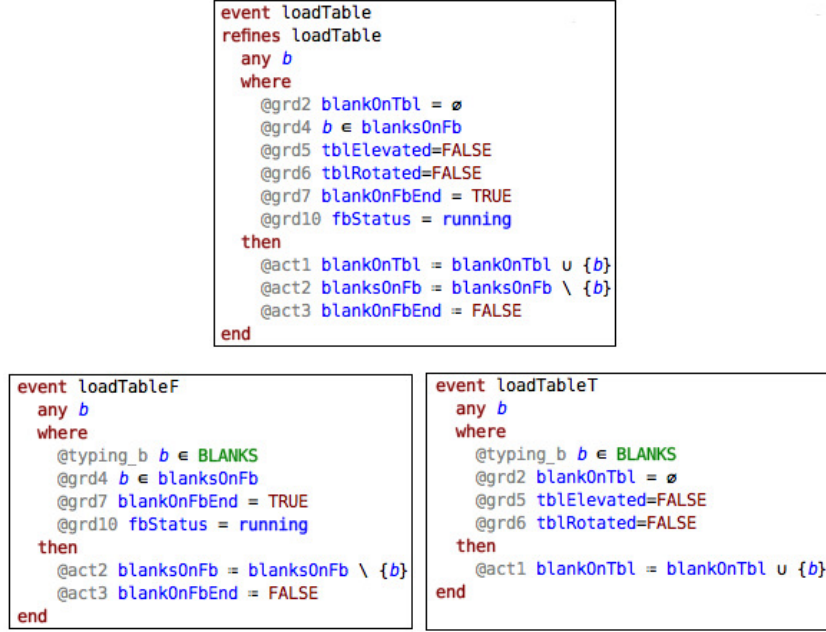


Figure 6.14: Event Splitting for SED

we had to do vertical refinement in order for us to perform SED unlike SVD where we only carried out horizontal refinements before the decomposition. So, it depends on the type of system being modelled and for distributed systems, the SED approach seems more appropriate.

After decomposing the model into sub-models, we could then refine each of these sub-models independently. In case of SVD, we had to maintain the restrictions of the SVD style while refining these sub-models, i.e., to ensure that the shared variables and external events were not refined. We further refined the ‘press’ sub-model vertically by introducing actuators and sensors. This involved another three levels of refinement and was done using a refinement pattern for control systems [But09e]. Other sub-models could also be refined similarly.

This gives us a product of PC which models a particular topology (see Figure 6.15) and how the physical components are connected to model the production cell plant. The figure is designed to visualise the topology in terms of events and helps in reuse for instantiation of alternative topologies. The events placed inside the boxes are local for the components and those placed between the boxes represent shared events containing topological information for modelling the connectivity of the components. This is an example of domain specific instance modelling with Event-B. Each different topology is an instance of PC product line and we can build more variants of PC by

Figure 6.15: PC Topology 1

Figure 6.16: PC Topology 2

selecting a different configuration (or topology) of these physical components. For example, if we want to model a production cell with two press components for processing blanks twice and using two robots. We can call this ‘topology 2’ where *robot1* picks blanks from the *table* and drops onto *press1* and *robot2* takes the blank from *robot1* which picks it from *press1* and drops on *press2* (see Figure 6.16). Here we are interested in exploring to what extent we can reuse the models of *topology1* while modelling *topology2* and hence the proof effort. For *topology2*, we had to do instantiation and refactoring to simply duplicate the functionality of existing components. This means that we would not have to reprove the models which have already been proved for *topology1*. This is because renaming of elements would not affect the POs and is currently supported by the refractory plug-in [Sil11b]. We only had to prove the POs generated for any additional information modelled in the second topology. For example, we specify that *arm1* of *robot2* collects the blank from *arm2* of *robot1* unlike picking it from the *table* in *topology1*.

Figure 6.17 shows the refinement architecture for modelling the two topologies and their components as achieved after decomposition. An example of event instantiation while modelling *topology2* after *topology1* is shown in Figure 6.18 where event *loadPress* is duplicated for the two presses (events: *loadPress1* & *loadPress2*). This type of instantiation and refactoring has no proof burden. The POs for *topology2* were discharged in the same way as *topology1*. Figure 6.19 shows the number of POs for both topology developments at different refinement levels and how these were discharged, i.e., automatically or interactively. The Figure would be more interesting if we could reflect the percentage of POs reused when modelling *topology2* after *topology1*. So, this shows that we can reuse the existing models and their proofs if we have tools to automate the instantiation and refactoring. Hence this exercise generated additional tooling requirements which are discussed later.

PC Controller-based: In the controller-based PC modelling, the functional requirements to model the behaviour of each controller were grouped together as a feature. So, the requirements specification was decomposed into various controller features. We also generalised the requirements for each of the controller so that we could model generic controllers which could then be specialised and reused for modelling various controllers of different physical components of PC. Hence, the controller-based modelling of PC was a result of decomposition plus generalization. Table 6.2 shows part of the requirements specification for the *table* feature of component-based PC and the *movement* feature for controller-based PC. This shows how we can define the feature in terms of requirements for two styles of modelling the PC while making the features more reusable. The compositional require-

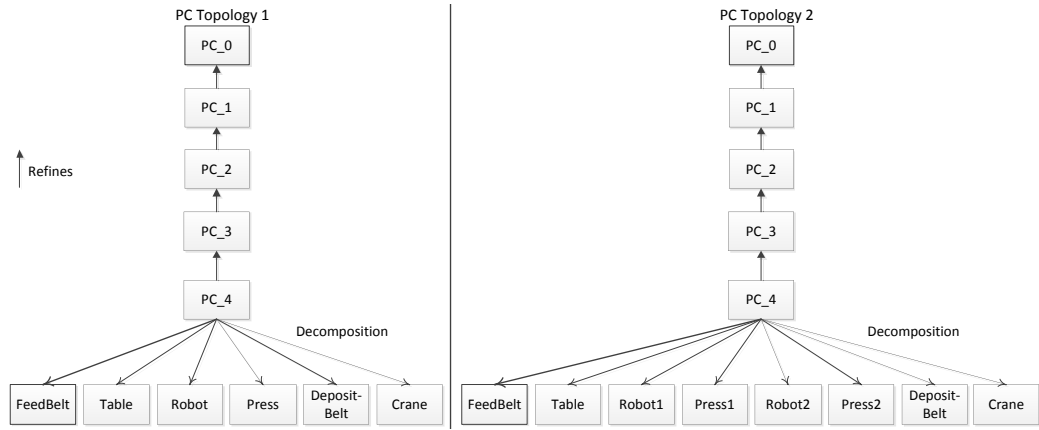


Figure 6.17: Refinement architecture for modelling the two topologies

```

event loadPress
  any b
  where
    @grd1 beBLANKS
    @grd2 pr ∈ ran(position)
    @grd3 bedom(position)
    @grd4 position(b) = arm1
    @grd5 pressPos=mid
    @grd6 robotPos=pos2
  then
    @act1 position(b) = pr
  end

event loadPress1
  any b
  where
    @grd1 beBLANKS
    @grd2 pr1 ∈ ran(position)
    @grd3 bedom(position)
    @grd4 position(b) = arm1r1
    @grd5 press1Pos=mid
    @grd6 robot1Pos=pos2
  then
    @act1 position(b) = pr1
  end

event loadPress2
  any b
  where
    @grd1 beBLANKS
    @grd2 pr2 ∈ ran(position)
    @grd3 bedom(position)
    @grd4 position(b) = arm1r2
    @grd5 press2Pos=mid
    @grd6 robot2Pos=pos2
  then
    @act1 position(b) = pr2
  end

```

Figure 6.18: Event instantiation example for PC topology2

	PC Topology 1				PC Topology 2			
Ref. level	Auto	Manual	Reviewed	Total	Total	Auto	Manual	Reviewed
PC-0	3	-	-	3	6	6	-	-
PC-1	36	1	-	37	56	55	1	-
PC-2	22	4	-	26	26	16	10	-
PC-3	41	4	-	45	59	55	4	-
PC-4	10	-	-	10	8	8	-	-
PC-5	84	-	-	84	190	188	2	-
PC-6	20	-	-	20	40	40	-	-
PC-7	188	-	8	196	432	401	11	20
Total	404	9	8	421	817	769	28	20

Figure 6.19: POs for two topology developments

ments are modelled while actually composing various components, this may include topological information and how components are connected together. The controller-based PC models consisted of *loader*, *movement*, *rotation* and *magnet* controllers. A member of PC product line could be modelled by instantiating and composing these controller-based reusable features. These features were then refined independently. We discuss the refinement of *magnet* and *movement* features below where we introduced sensors and actuators in various refinement steps using the pattern for refining control systems as suggested in [But09e].

Magnet Controller: At the abstract level, we have events for picking and dropping of blanks by a component. A component which has not already picked a blank can do so and a component which has picked a blank can drop it. The feature will be instantiated to a specific component such as a *crane* or a *robot arm*. The model is quite abstract and the details are added later in the refinements and during specialization. In the first refinement, we added sensor for magnet which informs the controller whether a blank has been picked up or dropped off. An electromagnet switch acts as an actuator for the magnet which performs the pick and drop of blanks. We have events for starting and stopping the magnet and switching the sensor on and off. In the second refinement, we differentiate between the actual and sensed values of the sensors. This is done to model the system closer to reality, as the actual value of the sensors at some point in time will be different from the sensed values. Similarly, in the third refinement, we refine the actuation where controller sets the actuation of the motor before the motor can be actuated.

Table 6.2: Requirements description for table and movement features of PC

<u>Table Component</u> - Move table upwards/downwards - Rotate table clockwise/anti-clockwise - Table must not rotate when it is at low position - Table must not move down if it is rotated (rotate backward first and then then move down) or if it is already not elevated - The table must not rotate clockwise if it is in a position to deliver blanks (unloading position) - The table must not rotate at all if it is not elevated <u>Compositional Requirements</u> - Drop blank on table from feed belt when it is in the loading position (not elevated and not rotated) - Robot picks blank from table when it is in unloading position (elevated and rotated)	<u>Generic Movement Controller</u> - Move a component from position A to position B and vice-versa <u>Instantiation Requirements</u> - Extract/Retract Arm1 - Extract/Retract Arm2 - Move Feed Belt/Deposit Belt - Move the Table upwards/ downwards - Move Press to upper/lower/middle position - Move Crane To and from Feed Belt/Deposit Belt <u>Compositional Requirements</u> - Extract/Retract Arm1 if robot is facing table or facing press while press is in middle position - Extract/Retract Arm2 if robot is facing deposit or facing press while press is in lower position - Table must not move down if it is rotated (rotate backward first and then move down) or if it is already not elevated - The press must not move downward if it is in lower position and must not move upward if it is in upper position - Crane should only move towards feed belt if it is positioned on deposit belt and vice-versa

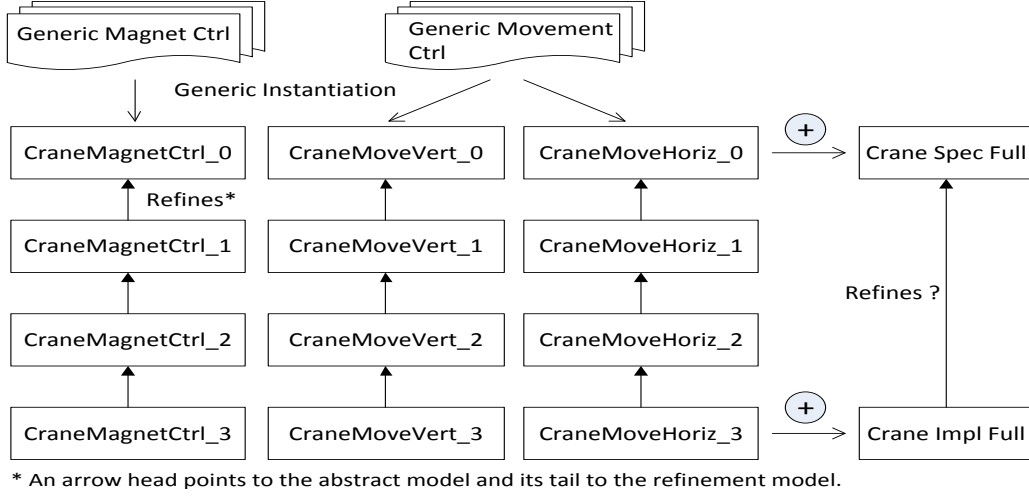


Figure 6.20: Crane Instantiation

Here we split the actuation events into two, i.e., an event for setting the actuation of magnet by the controller and the event for magnet to actuate accordingly.

Movement Controller: At the abstract level, we have events for moving a physical component forward and backward between two positions. The feature will be instantiated to a specific component such as a *press* or a *crane*. During the first refinement, we added sensors for the two positions and a motor for moving backward and forward. Events were added for starting and stopping the motor at different positions and switching the sensors on and off. In the second refinement, we differentiate between the actual and sensed values of the sensors as discussed earlier. Using the same refinement style, at third level of refinement, we differentiate between setting the motor's actuation by the controller from its actual movement.

Instantiation & Composition:

The *magnet* and *movement* controllers provide us refinement chains of generic Event-B models for the two features. In order to model any component of the PC, we need to instantiate and compose these chains of models. The composition is done in two phases. In the first phase, we compose instantiated models to build a component (e.g., *crane* or *robot*) and in the second phase, we compose all the components while providing topological information. For example, if we want to model the *crane* component, we have to specialize one instance of the *magnet* controller to pick and drop blanks and two instances of the *movement* controllers for moving the *crane* horizontally and vertically, as shown in Figure 6.20. In this example, we have three refinements in each development which align well during the composition. This

<pre> event PickBlank any b where @grd1 XcompX ∈ ran(position) @grd2 b ∈ dom(position) then @act1 position(b) = XcompX end </pre>	(before)	<pre> event CranePickBlank any b where @grd1 crane ∈ ran(position) @grd2 b ∈ dom(position) then @act1 position(b) = crane end </pre>	(after)

Figure 6.21: Event Specialization for Crane

alignment issue needs to be explored further to address the composition of Event-B developments having different number of refinements. Figure 6.21 shows a simple example where event *PickBlank* of *magnet* controller is specialized for the *crane* component. Here the generic model parameter *XcompX* is replaced by *crane* provided both of these are of the same type. For now, we use *X...X* as a syntactic convention to model a generic parameter, given that the current Rodin tool does not support generics.

The composition of abstract level models from each refinement chain would give us an abstract specification for the *crane*. We also had to do some guard strengthening and add some invariants during the composition. The composition of implementation level models for each refinement would provide us with the implementation of the *crane*. Again extra guards for events and invariants were needed. Figure 6.22 shows two events from *magnet* and *movement* controllers for picking up blanks by *crane* and the movement of *crane* towards *feed belt* (before composition). Figure 6.23 (after composition) shows these events with extra guards added during the composition. For example, *grd3* of *CranePickBlank* event specifies that the *crane* can only pick a blank when it is positioned on the deposit belt. Similarly, *grd2* of *moveToFB* event in Figure 6.23 specifies that the *crane* can only move towards the *feed belt* if it has picked up a blank. The guard *grd2* of *CranePickBlank* event means it can pick any blank in the system. When we finally compose all the components to model the entire PC in the second phase of composition, we will need to strengthen this guard to say that the *crane* can only pick a blank from the *deposit belt*. Here we would need to give topological information of PC in terms of how different components are connected to each other as discussed in the components-based PC example earlier.

We call this style of composition ‘feature composition’ where additional information can be added during the composition. As of yet, this style of composition does not guarantee refinement preservation between the composed abstract and implementation models (see ‘refines?’ in Figure 6.20). In

<pre> event CranePickBlank (before) any <i>b</i> where @grd1 ~~<i>crane</i> ∈ ran(position)~~ @grd2 ~~<i>b</i> ∈ dom(position)~~ then @act1 <i>position(b) = crane</i> end </pre>	<pre> event moveToFB (before) where @grd1 <i>cranePosHorz</i>=posDB then @act1 <i>cranePosHorz = posFB</i> end </pre>

Figure 6.22: Events of Magnet and Movement Controllers

<pre> event CranePickBlank (after) any <i>b</i> where @grd1 ~~<i>crane</i> ∈ ran(position)~~ @grd2 ~~<i>b</i> ∈ dom(position)~~ @grd3 <i>cranePosHorz = posDB</i> then @act1 <i>position(b) = crane</i> end </pre>	<pre> event moveToFB (after) where @grd1 <i>cranePosHorz</i>=posDB @grd2 ~~<i>crane</i> ∈ ran(position)~~ then @act1 <i>cranePosHorz = posFB</i> end </pre>

Figure 6.23: Guard strengthening of events during composition

order to deal with this kind of composition, we need support for proof reuse. By this we mean to find a way of automatically discharging composite POs with the help of already discharged POs of the components being composed. This requires further work. An alternative approach is discussed in [GPB11] for the ATM example where we can avoid reproof by following a modelling pattern. In comparison to the component-based approach discussed earlier, this style of modelling SPLs in Event-B seems more appropriate because it provides more reuse opportunities.

The shared-event composition could not be applied here due to the shared state between the components being composed. The shared-variable composition approach is too constraining and could only be used here if we start with an abstract model containing the functionality of both the *magnet* and *movement* features. We could then decompose these into two, refine each of these, instantiate for the crane and compose to build the crane model.

PC – Domain-specific SPL modelling through contexts: We also modelled a generic component-based PC which supports the two topologies mentioned earlier. The variability is provided through the context which means the machine for both the topologies will remain same and we could have a different topology by just switching the contexts. We modelled the topology of PC in the context, i.e., we specify in the context what are the physical components and how these could be connected to each other.

```

invariants
  @inv1 blankPosition ∈ BLANKS ↔ CPT
event loadComponent // load a blank on to a Component
  any b cpt
  where
    @grd1 b ∈ BLANKS
    @grd2 cpt ∈ CPT
    @grd3 cpt ∉ ran(blankPosition) // cpt is vacant
  then
    @act1 blankPosition(b) = cpt
  end
event passBlankBwCpts // pass a blank from one to another component
  any b cpt1 cpt2
  where
    @grd1 b ∈ BLANKS
    @grd2 cpt1 ∈ CPT
    @grd3 cpt2 ∈ CPT
    @grd7 cpt1 ↦ cpt2 ∈ cptGraph
    @grd4 b ∈ dom(blankPosition)
    @grd5 blankPosition(b) = cpt1 // blank is on cpt1
    @grd6 cpt2 ∉ ran(blankPosition) // cpt2 is vacant
  then
    @act1 blankPosition(b) = cpt2
  end
end

constants fb tbl arm1 arm2 pr db cr cptGraph
sets CPT POS
axioms
  @axm1 partition(CPT, {fb}, {tbl}, {arm1}, {arm2}, {pr}, {db}, {cr})
  @axm3 cptGraph ∈ CPT ↔ CPT
  @axm4 cptGraph = {fb ↦ tbl, tbl ↦ arm1, arm1 ↦ pr, pr ↦ arm2, arm2 ↦ db, db ↦ cr}

```

Figure 6.24: Example of domain-specific modelling through context

The machine is modelled in a generic way, e.g., an event *loadComponent* is used to model the loading of blanks onto a PC component and an event *passBlankBtwCpts* for moving a blank from one component to another. This is shown in Figure 6.24. The invariant *inv1* defines the position of blank at a component. The topology is given in the contextual part at the bottom where *cptGraph* defines the components graph, i.e., valid ways of connecting different components. This is then used in *grd7* of the *passBlankBtwCpts* event to make sure that the blank is passed between connectable components.

The advantage of modelling in this way is that we will not need to reprove a variant of PC resulting from static variability through context switching. The disadvantage is that the modeller may not visualise various events of the machine for a particular topology as this machine may not be instantiated. This could be a useful domain modelling activity for exploring variability of a product line in a distributed environment.

Evaluation: By modelling the PC in three different ways, we have explored to what extent we can use existing Event-B tools and techniques for feature-based product line modelling. This also enabled us to figure out the requirements for future tooling and techniques (discussed later) that can

further facilitate such development approach to benefit from reuse of existing models and their proofs.

The first style of modelling – component-based – is a natural approach of modelling in Event-B which used both types of decomposition techniques to reduce the complexity of modelling and proving by decomposing large models into smaller sub-models. It also showed that we could model variants of PC product line by configuring the different topology of physical components of PC. This is not a typical example for feature-based modelling as the entire model needs to be redone again for building a second topology after doing the first one. But this allowed us to explore requirements for instantiation and refactoring tool support that could be useful in automating this approach.

The second approach of controller-based modelling is more feature-oriented as we have modelled generic reusable features that could be instantiated and composed in different ways to model different PC components and hence benefit from their reuse.

The third approach of modelling static variability through context switching allows to evaluate the scope of a product line and without doing proving effort upfront. This in another way of modelling component-based PC and suits the product line development approach as we can figure out the common base for different variants (topologies) of the PC, and the configuration or the variability is embedded in the context. This could be useful to foresee how a product line would evolve for a particular domain and later on this could be modelled in one of the two styles mentioned to build a database of reusable features.

6.5.5 Current Feature Modelling Tool Support for Event-B

Based on our feature modelling notation discussed in Section 6.5.3, we have developed a feature modelling tool to specify feature models of product lines and to configure the feature models to generate their instances. The tool is open source, developed in Eclipse using Java and integrated as a plug-in to the Rodin platform. It consists of two parts, i.e., feature model editor and configuration editor, discussed next.

Feature Model Editor

Our feature modelling tool includes a graphical feature model editor developed using GMF (Graphical Modelling Framework) [Onl11]. This editor can be used to draw feature models for product lines in a free form tree structured hierarchy and uses our graphical notations of feature modelling. There is a

three way validation mechanism provided by the editor to ensure that valid feature models are drawn which could then be configured. The division is based on how this is implemented by the tool. Firstly, feature models should conform to the feature metamodel. Secondly, it does not let the user draw a feature model that *violates* any of the following validation cases:

- A feature model may not have cycles, i.e., a feature x having a child feature y , which is a parent feature of x .
- A feature may not include and exclude the same feature.
- An Event-B feature must be a leaf feature in the feature model tree and should not have further features or groups.

Thirdly, the editor validates the feature model when it is saved based on the following validation cases and warns the user if any of the cases is violated by the model. This differs from the above as it lets the user save an inconsistent/incomplete model to be completed later. Note that we have not yet explored automated feature model verification and the generated instance validation as proposed by Sun et al. [SZW05].

- A feature x may not include features y and z , which exclude each other. If this scenario is present in the feature model, it is not possible to generate a valid instance of the feature model.
- A feature may not exclude any of its ancestor features.
- A feature may not be unreachable during the configuration or may not be selected in any valid configuration. An example would be a group of two features with group cardinality ‘1..1’ and one of the features is mandatory. In this case, the user will never be able to have the optional feature selected in a valid configuration as doing that will violate the group cardinality due to the other mandatory feature.
- A feature model may not have orphan features.

A metamodel provides a language or notations for building the models. The feature models built using the feature model editor are transformed into their equivalent EMF metamodels at run-time and for each product family. This model to metamodel transformation is needed in order to instantiate the feature models and this forces the instances to conform to their metamodel. So, a feature model is an instance of the metamodel defining our feature modelling notations and it also serves as a metamodel for any of its instances.

The transformation is done using Epsilon Transformation Language (ETL) [KPP08]. After transforming feature models into metamodels, for different product lines, these are then used as an input to the feature configurator for instantiation discussed next. This model transformation process is internal to the tool and not visible to the modeller. This is shown in Figure 6.25 as part of the feature modelling tool architecture.

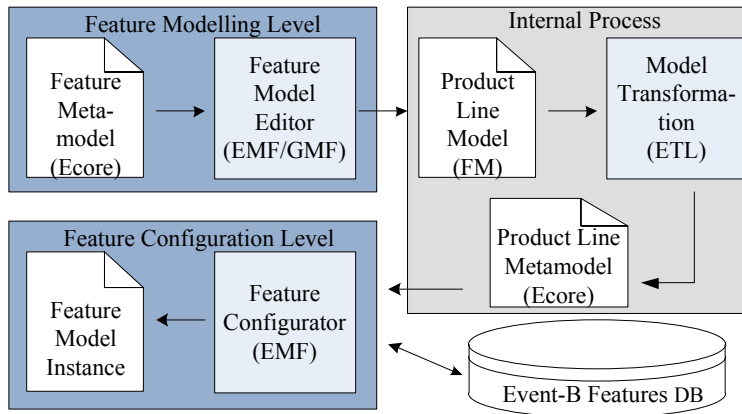


Figure 6.25: Feature Modelling Tool Architecture

Feature Configurator

The feature configurator is a collapsible tree-structured editor (screenshot in Figure 6.26) that allows the user to configure a feature model by selecting features that they want to include in a particular product. The editor then highlights any conflicts and provides options for resolving these conflicts automatically/interactively. The selected features (Event-B models) are then composed into a single composite Event-B model. The configurator enforces some of the constraints provided in the feature model. It automatically selects the mandatory features and highlights any violation of cardinality constraints. Whenever a feature is selected, the tool automatically selects/deselects features specified using includes/excludes constraints. The configurator also shows Event-B elements of machines and contexts (e.g., variables, events etc.) for the Event-B features which is different to other feature modelling tools (e.g., FeatureHouse [AKL09]).

At the moment it detects naming conflicts within Event-B models (e.g. variables or events having same name). It provides ways to automatically resolve these name clashes either by making them disjoint through refactoring or by simply deselecting repeating entries in multiple features. It also helps the user in automatically selecting any dependencies, for example, if an event

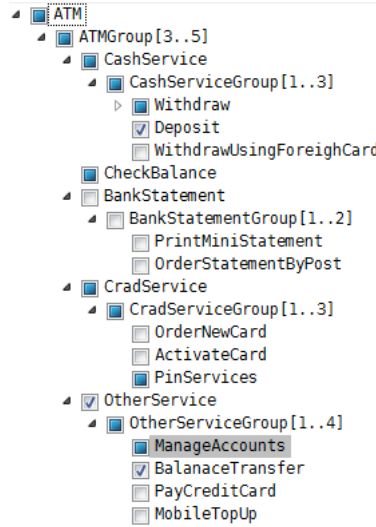


Figure 6.26: Feature Configurator Screenshot

is selected, it can then select the related variables and their invariants to build the correct model.

Once all the desired features are selected and conflicts are resolved, these are composed to generate a composite Event-B feature. All the machines are merged into a machine and all the contexts are merged into a context. This can be called as a structured cut-and-paste composition. The editor also enables the user to merge multiple events into a single event. This concatenates the actions and conjoins the guards to maintain invariant preservation. This composition of Event-B features into a composite feature is required in order for us to reason about the complete model of the generated instance, e.g., using animation, theorem provers etc.

The feature modelling tool was quite useful while modelling our case-study examples as we had to compose models in different ways to experiment with our modelling approaches and to use existing decomposition techniques, specifically SVD, since there is no tool support for shared-variable composition. The tool will be extended further to include the requirements we have generated during this research, presented in the next Section.

6.5.6 Guidelines for Feature Modelling in Event-B & Tooling Requirements

Our case-study experiments reveal some patterns of modelling that could be used as a guideline for specifying SPLs in Event-B. These patterns make use of Event-B's existing tools and techniques. If we could develop further tool

support as discussed later, then this whole process could be automated saving lot of user time and effort. At the moment, the lack of proper tool support meant that we had to do lot of things manually, e.g., trivial refactoring and instantiation of a generic model into two disjoint models. In the following we present some guidelines for formal feature-based modelling in Event-B which generalise our approach of modelling the two case-studies discussed earlier.

By grouping a list of requirements into small features as we did for controller-based PC modelling, making each feature as a stand alone generic development, we can reuse and specialise these features in different configurations to build variants of a PL. This could also be done by first modelling the system up to a few refinements as a whole and then trying to figure out how the system could be decomposed in terms of requirements/functionality. In this way, the modeller will be able to identify earlier on what sort of information would be required in order to compose these features so that the generic features could be modelled with that input/output connector information as in component-based software development. This would serve as a domain-analysis step to predict reusability of features before actual modelling them as reusable features.

The decision to use top-down usual modelling approach of Event-B, bottom-up approach of SPLE or the middle-in approach of the ATM example [GPB11] would vary for different domains. The domain expert or system modeller would be in a better position to judge which approach would suit more for a particular system.

Based on experience of the two case-studies modelled in different ways, if the user is planning to model a PL and would need to build many variants of the PL in future, then its probably useful to take the middle-in approach and make the features as generic as possible. If the abstract models of the features being modelled initially could be considered as a result of SVD, then their recomposition at any later refinement will be correct by construction. These features later on could also be decomposed using SED for architectural decomposition and that would not violate the correctness promise of SVD if the same pattern of modelling is followed. If this pattern is not followed and feature are modelled as generic reusable models independently and later on composed and additional composition time information is supplied (discussed as Feature Composition earlier), then the user need to reprove the composite model. This could only be helpful if we have a mechanism for proof reuse that would reduce the reproving effort.

Also, if the features being developed can use generic placeholders which could be filled during composition, then that would reduce effort of reproving by simple refactoring. The generic placeholder could the be specified, e.g., $X...X$ at the moment as Event-B tool will not raise an error, so the compo-

sition tool would force the user to fill these placeholder at the composition time.

Future Tool Support Requirements

In order to make use of the feature modelling approach suggested, the following tool support will be needed to facilitate the modeller.

- **Refactoring Support:** During the composition of features, allow the user to add new variables, invariants and guards. These additional variables, guards and invariants should be static-checked for errors and instantly reported to the user. The user should be able to guide the refactoring, i.e., when refactoring a machine, allow the user to suggest prefixes. This should also support instantiation of generic refinement chains by refactoring. The refactored elements should also be static-checked for any errors. The tool should provide support for generic placeholders. For example, an element name (e.g., vars, events etc.) must be provided by the user at the time of composition when a model contains any placeholder (e.g., X...X).
- **Feature Modelling & Topology Tool:** The tool should allow the user to specify cardinality for feature instantiation (cloning), so that a feature could be replicated a number of times in a particular instance. For example, in PC feature model, the feature ‘press’ could have a cardinality ‘1..n’, which means any instance of the PC could have at least one and up to ‘n’ presses. This must be then supported in the configuration editor and would also need user guided refactoring support as discussed above.

The user should be able to annotate the feature model, i.e., to specify composition rules, more complex constraints and other information that they think might be helpful during the configuration process. For example, the composition rule and rationale provided in the example feature model of Figure 6.9. This would help the modeller during the selection of a particular feature while instantiating a product line.

After drawing a feature model, allow the user to draw a topology graph based on that feature model to visualize the feature model instance and how different features will be connected to each other in the graph, as discussed in PC. The features are drawn as nodes and the user should be able to select the transitions between these nodes by selecting an event from the available features in the feature machine. This may be also be achieved by adding extra constraints to the feature model that specifies

connection between components. So the user specifies which component is connected to which other components and this information could be used to draw transitions between different nodes. At the moment, the feature modelling tool only supports two constraints, i.e., includes and excludes.

- **SVC Tool:** There should be a tool for composing models using the SVC style. This may also include support to automatically generate external events and shared-variables so that the models could be composed using shared-variable composition style. Also, it would be helpful for the modelled to have a single tool that could be used to compose models using any of the composition styles, i.e., SEC, SVC, fusion and feature composition.
- **Composition Replay Support:** All the composition decisions must be saved so that the user can come back later to modify or redo the composition without losing additions or refactoring etc. It would be useful to have a script of all the configuration decisions so that the user can replay the configuration by editing the script (may be doing some refactoring etc.). This will be like an audit trail of all the actions performed during the configuration. This may also include graphical representation of the actions performed during the composition to visualise more complex multi-step compositions.

6.5.7 Conclusion & Future Work

We have given an overview of our research on feature-oriented reuse approach using Event-B and to what extent this is possible using existing Event-B's tools and techniques. Although, it seems possible to model SPLs using existing techniques with some restrictions, we still need more tool support to gain full benefits of formal SPL engineering. Our case-study examples have revealed some patterns of modelling that could be used as a guideline for SPLE using existing (de)composition approaches and to minimise the proof effort while reusing already proven Event-B features. We have also generated some tooling requirements that could facilitate the user for feature-modelling in Event-B. Some interesting research questions came up during this work that would need to be addressed in the future. For example, to what extent our suggested modelling pattern could be generalised. This would require further work where we could apply these guidelines and pattern on more complex case-studies.

We also need to do some research on a proof reuse approach that would allow more complex and freer form of composition with minimal proof burden

for the modellers. We would like to see if the proof composition approach of [TSKA11] can be applied to Event-B using the case-studies presented. The issue of feature interactions [CKE03] also needs to be explored for feature-oriented modelling in Event-B.

Chapter 7

Modelling Hardware Architectures

*M. Daneshtalab, M. Kamali, L. Petre, Y. Prokhorova, K. Sere,
E. Troubitsyna, L. Tsiopoulos*

7.1 Refinement-Based Modelling of 3D NoCs

7.1.1 Introduction

The Network-on-Chip (NoC) architecture paradigm, based on a modular packet-switching mechanism, can address many of the on-chip communication design issues such as performance limitations of long interconnects and the integration of high numbers of Intellectual Property (IP) cores on a chip. However, the 2D-chip fabrication technology faces many challenges in the deep submicron regime even when employing NoC architectures, e.g, the design of the clock-tree network for large chips, limited floor-planning choices, the increase of both the wire delay and power consumption, the integration of various components that are digital, analog, MEMS and RF, etc. Three Dimensional Integrated Circuits (3D ICs) have been emerging as a viable candidate to achieve better performance and package density as compared to traditional Two Dimensional (2D) ICs. In addition, combining the benefits of 3D ICs and NoC schemes provides a significant performance gain for 3D architectures [FP09, PED⁺08, LB10].

Three dimensional Networks-on-Chip (3D NoCs) [FP09] provide more reliable interconnections due to the increased number of links between com-

ponents. Due to their promise of parallelism and efficiency, 3D NoCs have a critical role in leading towards reliable computing platforms. However, the majority of their evaluation approaches are simulation-based tools, such as XMulator [NMSSA07], Noxim [PHKC09], etc. Simulation-based approaches are usually applied in the late stages of design and are limited, e.g., by the length of time that a system is simulated. This means that exhaustive checking of all the system states is impossible in practice for complex 3D NoCs and thus, simulation is not suitable for verifying the correctness of a NoC design.

Another approach to address this problem is via formal methods. Prominent examples of applying formal methods are provided by, e.g., Intel [Har03, KGN⁺09] and IBM [LH03] for formally verifying hardware or systems-on-chip (SoC) [GSLG04]. By using rigorous mathematical techniques, it is possible to deliver *provably correct* systems. In this report, we propose a top-down formalisation of the early 3D NoC design. We specify here the general structure of a 3D NoC at a high level of abstraction in Event-B. The specification formulates the main constraints of the communication model, needed to prove its correctness. Our definition for correctness at this abstract level of modelling is to show that a package injected in the network is eventually received at the destination. We propose three different abstract models M_0 , M_1 , and M_2 for a 3D NoC so that $M_0 \sqsubseteq M_1 \sqsubseteq M_2$, where ‘ $\sqsubseteq$ ’ denotes the refinement relation. Furthermore, each of these models can be refined into more concrete models to define specific 3D NoC designs in the early stages of the system development. When the concrete models preserve the correctness properties of the abstract models, we guarantee the correctness of the concrete 3D NoC designs. As an application of the general 3D NoC designs, we model the XYZ routing algorithm by refining the M_2 abstract model. To verify the XYZ routing algorithm, we generate the proof obligations using the Rodin tool and discharge them automatically or interactively.

The contribution of this section is based on the conference publication [KPSD11]. We proceed as follows. In Section 7.1.2 we propose three increasingly more detailed formal models for a 3D NoC together with the constraints for proving correctness. In Section 7.1.3 we illustrate the formal modelling of the XYZ routing algorithm as a case study. In Section 7.1.4 we discuss the proof obligations, while in Section 13.4.6 we present concluding remarks.

7.1.2 Three Abstract Models for the 3D NoC: M_0 , M_1 , M_2

In this section we formally develop three high-level models M_0 , M_1 , and M_2 for the 3D NoC. Our models are at three increasing levels of detail so that each model is a refinement of the previous one: $M_0 \sqsubseteq M_1 \sqsubseteq M_2$. In the initial model, we specify a network of nodes and define the correctness properties of this network based on a specific data structure called *pool*, as suggested by [Abr10]. In the second model, we add new data and events to model the 3D mesh-based NoC architecture; besides, we specify the channels between nodes. In the third model, we model buffers for nodes and refine the communication model.

By starting from an initial model that is rather abstract, i.e., without detailing the communication topology, we obtain a rather general starting point that can later be refined to various topologies. Moreover, adding channels and ports only in the second model leads to a clean modelling of the basic communication mechanism (via routing and switching) in the initial model; the required detail (of channels and ports) are not needed for understanding the communication mechanism. Adding buffers in the third model illustrates an extra level of detail. Networks where the nodes have no buffers for communication will, therefore, employ the second model as their abstraction and not the third.

The initial model M_0

The first model M_0 that we construct is rather abstract: we do not consider the numerous parts of the network such as channels or buffers; they will be introduced in subsequent refinements. M_0 will thus allow us to reason about the system very abstractly [Abr10]. The model M_0 is formed of the static part and the dynamic part, as follows.

The Static Part. The static part of our model is described in Fig. 7.1 and contains the sets *MESSAGES*, *ROUTER*, *DATA* and the constants *data*, *des*, *src* and *Neigh*. The message identifiers are modeled by the non-empty and finite *MESSAGES* set. We use the following modelling idea for messages. A message id in the *MESSAGES* set relates to a triple (*data*, *source*, *destination*) where *data* is an element of the *DATA* set, *source* models the source node where a message is injected, and *destination* models the destination node where a message should be received. A message should not be destined to its source node. The set of network nodes and data are modelled by the sets *ROUTER* (finite and non-empty) and *DATA* (finite and non-empty),

respectively. The relation *Neigh* (non-empty, symmetric, and non-reflexive) models the neighbour structure i.e., which node can communicate with which node.

SETS	MESSAGES	ROUTER	DATA	
CONSTANTS	data	des	src	Neigh
AXIOMS	$MESSAGES \neq \emptyset \wedge \text{finite}(MESSAGES)$ $ROUTER \neq \emptyset \wedge \text{finite}(ROUTER)$ $DATA \neq \emptyset \wedge \text{finite}(DATA)$ $data \in MESSAGES \rightarrow DATA$ $src \in MESSAGES \rightarrow ROUTER \wedge des \in MESSAGES \rightarrow ROUTER$ $\forall m, sp, dp \cdot m \in MESSAGES \wedge sp \in ROUTER \wedge dp \in ROUTER$ $\wedge m \mapsto sp \in src \wedge m \mapsto dp \in des \Rightarrow sp \neq dp$ $Neigh \in ROUTER \leftrightarrow ROUTER$ $Neigh \neq \emptyset \wedge Neigh = Neigh^{-1} \wedge dom(Neigh) \triangleleft id \cap Neigh = \emptyset$			

Figure 7.1: M_0 : the static part

To define structure types such as records in Event-B, we use functions to represent attributes. Therefore, our modelling idea translates to the functions *data*, *src* and *des* with ranges *DATA*, *ROUTER*, and *ROUTER*, respectively.

The Dynamic Part. In our network model we use the following condition for modelling the communication correctness: the messages in the network will eventually reach their destinations. For this, we define two message subsets and one partial message-to-node map as machine variables: *sent_pool* $\subseteq MESSAGES$, *received_pool* $\subseteq MESSAGES$ and *moving_pool* $\in sent\_pool \text{ functional } ROUTER$.

The *sent_pool* subset denotes the list of messages injected into the network. The *sent_pool* subset is updated whenever a new message is injected into the network, while the *moving_pool* subset denotes the current position of traveling messages. All the messages injected into the network are added to the *moving_pool* and whenever a message is routed from a node to another one, the current position of that message is updated in the *moving_pool*. The *received_pool* subset denotes the list of messages received from the network by destination nodes. Whenever a message is received at its destination, it will be added to *received_pool* and removed from *moving_pool*. The behaviour of message pools is illustrated in Fig. 7.2.

To model the communication and the message pool functions, we define three events as explained below. The *sent_message* event described in Fig. 7.3a handles the injection of a new message into the network. Whenever a message is injected into the network both *sent_pool* as well as *moving_pool* are updated.

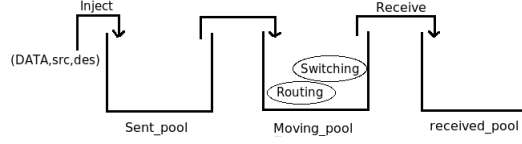


Figure 7.2: Message Pools

```

Event sent_message  $\hat{=}$ 
any
    current_msg
where
    current_msg  $\in$  MESSAGES
    current_msg  $\notin$  sent_pool
then
    sent_pool := sent_pool  $\cup$  {current_msg}
    moving_pool := moving_pool
     $\cup$  {current_msg  $\mapsto$  src(current_msg)}
end

```

(a) Message Injection

```

Event routing  $\hat{=}$ 
begin
    skip
end

Event switching  $\hat{=}$ 
Status anticipated
any
    current_msg new_position
where
    current_msg  $\in$  dom(moving_pool)
    new_position  $\neq$  src(current_msg)
    des(current_msg)  $\neq$  moving_pool(current_msg)
    new_position  $\mapsto$  moving_pool(current_msg)  $\in$  Neigh
then
    moving_pool(current_msg) := new_position
end

```

(b) Routing and Switching

Figure 7.3: M_0 Events

A message in *moving_pool* should be routed toward its destination. This is composed of two actions, one for deciding which node would be the next one (routing) and the other for transferring the message to that node (switching). These two actions are modeled respectively by the *routing* and *switching* events shown in Fig. 7.3b. In this abstract model we do not have any routing decisions, hence, the *routing* event is modelled by *skip*. The *switching* event in the M_0 model only transfers a message from the current node to one of its neighbours nondeterministically and updates the *moving_pool* by changing the current position of a message. To avoid cycling, we do not allow a message to return to its source. The reason for not considering a specific routing algorithm is that it makes our initial model more general and reusable for a wide variety of routing algorithms implementations. The *switching* event has the status *anticipated*.

The *received_message* event shown in Fig. 7.4a adds a message received at its destination to *received_pool* and removes the message from *moving_pool*. This event is convergent: if new messages are not injected to the network for a certain time, all the messages will be received at their destinations. This

```

Event   received_message  $\hat{=}$ 
Status convergent
any
    current_msg
where
    current_msg  $\in \text{dom}(\text{moving\_pool})$ 
     $\text{des}(\text{current\_msg}) = \text{moving\_pool}(\text{current\_msg})$ 
then
     $\text{moving\_pool} := \{\text{current\_msg}\} \Leftarrow \text{moving\_pool}$ 
     $\text{received\_pool} := \text{received\_pool} \cup \{\text{current\_msg}\}$ 
end

```

(a) Received-message Event

```

INVARIANTS
 $\text{dom}(\text{moving\_pool}) \subseteq \text{sent\_pool}$ 
 $\text{received\_pool} \cap \text{dom}(\text{moving\_pool}) = \emptyset$ 
 $\text{sent\_pool} = \text{received\_pool} \Leftrightarrow \text{moving\_pool} = \emptyset$ 
 $\forall \text{msg} \cdot \text{msg} \notin \text{sent\_pool} \Rightarrow \text{msg} \notin \text{received\_pool}$ 
 $\text{sent\_pool} \setminus \text{dom}(\text{moving\_pool}) = \text{received\_pool}$ 
 $\text{sent\_pool} \setminus \text{received\_pool} = \text{dom}(\text{moving\_pool})$ 

```

(b) Invariants (Pool Modelling)

Figure 7.4: M_0 Events and Invariants

is proved by means of the $(\text{sent\_pool} \setminus \text{received\_pool})$ variant denoting the difference between the sets sent_pool and received_pool .

In order to prove the communication correctness, we need to prove that the sent_pool subset eventually becomes equal with the received_pool subset and the moving_pool subset is empty when all the messages are received at their destinations. These properties are formulated in Fig. 7.4b as invariants.

M_0 is a general specification of a general network and will be refined to model 3D NoC communication designs in the following. Moreover, the model provides the necessary properties that should be preserved by refinement. These properties, that guarantee the overall communication correctness, are defined as the list of invariants.

The Second Model M_1

Transferring a message from a node to its neighbour in the model M_0 is achieved simply by copying the message from a node to another. In this section we refine the initial model M_0 to also specify channels specific to the 3D NoCs. To specify channels, we need a 3D NoC architecture. There are a num-

ber of 3D NoC architectures, e.g., mesh-based [FP09], tree-based [GPIS04]. We consider here NoCs with 3D mesh topologies. The 3D mesh-based NoC (Fig. 7.5(a)) consists of $N = m * n * k$ nodes; each node has an associated integer *coordinate triple* (x, y, z) , $0 < x \leq m$, $0 < y \leq n$, $0 < z \leq k$.

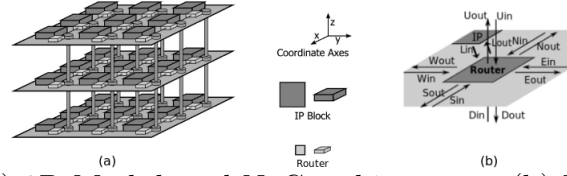


Figure 7.5: (a) 3D Mesh-based NoC architecture (b) Router channels

Our 3D NoC architecture employs seven-port routers: one port to the IP block, one port to above and below routers, and one in each cardinal direction (North, South, East and West), as shown in Fig. 7.5(b).

The Static Part. We extend the static part of the initial model M_0 in three ways: we map routers to coordinate triples, we add new properties for the *neigh* relation based on the coordinate triples, and we model ports and channels for the 3D NoC. In order to map routers to the coordinate triples, we define four constants: *coordX*, *coordY*, *coordZ* and *mk_position* as shown in Fig. 7.6. The *coordX*, *coordY* and *coordZ* constants represent coordinate triples (x, y, z) and the *mk_position* constant is a map associating each router to a position in space given by the coordinates. The *crossbarX*, *crossbarY* and *crossbarZ* constants model the number of nodes in X, Y and Z coordinate in the network, respectively.

Two nodes with coordinates (x_i, y_i, z_i) and (x_j, y_j, z_j) are connected by a communication channel if and only if $|x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 1$. To model this neighbour structure, the *Neigh* relation in the initial model M_0 is restricted in this model by adding the axiom in Fig. 7.7.

We define the *CHANNEL* set to model the communication channels between routers and we define the *PORTS* set to define the input and output ports of nodes in the static part of the second model. To show how two neighbours are connected to each other through channels, we define the *def_channel* and *mk_channel* relations with the help of axioms, as shown in Fig. 7.7. The *def_channel* relation models the relation of a port of a node to the corresponding port of its neighbour and the *mk_channel* relation maps the port relations to channels.

East and west ports of neighbour nodes with different X coordinate are related to each other through a channel. For instance, as shown in Fig. 7.8, *Ein* and *Eout* ports of node $(1, 1, 1)$ are connected to *Wout* and *Win* ports

SETS	<i>CHANNEL</i>	<i>PORTS</i>
CONSTANTS	<i>coordX</i>	<i>coordY</i> <i>coordZ</i> <i>mk_position</i>
	<i>crossbarX</i>	<i>crossbarY</i> <i>crossbarZ</i> <i>mk_channel</i>
AXIOMS	$ \begin{aligned} & \text{crossbarX} \in \mathcal{N}_1 \wedge \text{crossbarY} \in \mathcal{N}_1 \wedge \text{crossbarZ} \in \mathcal{N}_1 \\ & \text{mk_position} \in (1 \dots \text{crossbarX}) \times (1 \dots \text{crossbarY}) \times (1 \dots \text{crossbarZ}) \mapsto \text{ROUTER} \\ & \text{coordX} \in \text{ROUTER} \rightarrow (1 \dots \text{crossbarX}) \\ & \text{coordY} \in \text{ROUTER} \rightarrow (1 \dots \text{crossbarY}) \\ & \text{coordZ} \in \text{ROUTER} \rightarrow (1 \dots \text{crossbarZ}) \\ & \forall xx, yy, zz \cdot xx \in 1 \dots \text{crossbarX} \wedge yy \in 1 \dots \text{crossbarY} \wedge zz \in 1 \dots \text{crossbarZ} \\ & \quad \Rightarrow \text{coordX}(\text{mk_position}(xx \mapsto yy \mapsto zz)) = xx \\ & \quad \wedge \text{coordY}(\text{mk_position}(xx \mapsto yy \mapsto zz)) = yy \\ & \quad \wedge \text{coordZ}(\text{mk_position}(xx \mapsto yy \mapsto zz)) = zz \\ & \forall pos1, pos2 \cdot pos1 \in \text{ROUTER} \wedge pos2 \in \text{ROUTER} \wedge pos1 \neq pos2 \\ & \quad \Rightarrow \text{coordX}(pos1) \neq \text{coordX}(pos2) \vee \text{coordY}(pos1) \neq \text{coordY}(pos2) \vee \\ & \quad \text{coordZ}(pos1) \neq \text{coordZ}(pos2) \end{aligned} $	

Figure 7.6: M_1 : Static Part 1

AXIOMS
$ \begin{aligned} & \forall r1, r2 \cdot r1 \mapsto r2 \in \text{Neigh} \Leftrightarrow \text{abs}(\text{coordX}(r1) - \text{coordX}(r2)) + \text{abs}(\text{coordY}(r1) \\ & \quad - \text{coordY}(r2)) + \text{abs}(\text{coordZ}(r1) - \text{coordZ}(r2)) = 1 \\ & \text{def_channel} \in (\text{ROUTER} \times \text{PORTS}) \rightarrow (\text{ROUTER} \times \text{PORTS}) \\ & \text{partition}(\text{PORTS}, \{\text{Ein}\}, \{\text{Eout}\}, \{\text{Win}\}, \{\text{Wout}\}, \{\text{Nin}\}, \{\text{Nout}\}, \{\text{Sin}\}, \{\text{Sout}\} \\ & \quad , \{\text{Uin}\}, \{\text{Uout}\}, \{\text{Din}\}, \{\text{Dout}\}, \{\text{Lin}\}, \{\text{Lout}\}) \\ & \text{mk_channel} \in \text{def_channel} \mapsto \text{CHANNELS} \end{aligned} $

Figure 7.7: M_1 : Static Part 2

of node $(2, 1, 1)$ through a channel $((1, 1, 1) \mapsto \text{Eout}) \mapsto ((2, 1, 1) \mapsto \text{Win})$ and $((2, 1, 1) \mapsto \text{Wout}) \mapsto ((1, 1, 1) \mapsto \text{Ein})$ relations in *def_channel*. This connection of the ports of the neighbouring nodes on the X coordinate is modelled by the axiom shown in Fig. 7.9. The port relation between neighbours on other coordinates is defined by similar axioms which are not shown here due to lack of space.

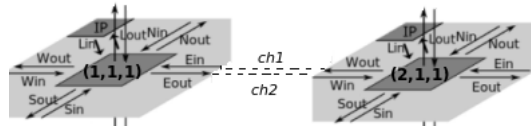


Figure 7.8: Channels in 3D Mesh-Based NoCs

The Dynamic Part. In the static part of the model M_1 , we define the 3D mesh NoC architecture with the triple coordinate of nodes and their channels. In the dynamic part of the model M_1 , we refine the dynamic part of the model

M_0 to specify the transferring of data through the communication channels, so that the overall correctness of communication holds.

The communication channels between routers are considered asynchronous channels, transferring data upon request. Each channel propagates data as well as control values. In our case, a control value models the fact that a channel is occupied by a message. When a message is injected to a channel, the control value of that channel is set to *busy* and when the message is received at the other side of channel, the control value of that channel is set to *free*.

AXIOMS

$$\begin{aligned}
& \forall n, m, i, j \cdot (n \mapsto i) \mapsto (m \mapsto j) \in \text{def\_channel} \wedge i = \text{Wout} \wedge j = \text{Ein} \\
& \Leftrightarrow \text{coordX}(n) - \text{coordX}(m) = 1 \wedge \text{coordY}(n) = \text{coordY}(m) \wedge \text{coordZ}(n) = \text{coordZ}(m) \\
& \forall n, m, i, j \cdot (n \mapsto i) \mapsto (m \mapsto j) \in \text{def\_channel} \wedge i = \text{Eout} \wedge j = \text{Win} \\
& \Leftrightarrow \text{coordX}(n) - \text{coordX}(m) = -1 \wedge \text{coordY}(n) = \text{coordY}(m) \wedge \text{coordZ}(n) = \text{coordZ}(m)
\end{aligned}$$

Figure 7.9: M_1 : Static Part 3

In order to model the transferring of messages through the communication channels, the variables *channel_state* and *channel_content* are defined in the second model to represent the control and the data value on each channel. Each channel can have the *busy* or *free* state. When the channel receives data, its state switches from *free* to *busy* and the message is added to the *channel_content*. When the channel transfers data to the end, *channel_state* changes to *free* and the channel is released by removing the message from *channel_content*. The invariants of M_1 model that, when a channel is released, then its content is empty and can thus receive the next message; when a channel is busy, the message is in the channel. We illustrate these invariants in Fig.7.10.

VARIABLES

channel_content *channel_state*

INVARIANTS

$$\begin{aligned}
& \text{channel\_content} \in \text{CHANNELS} \text{ injective } \text{MESSAGES} \\
& \text{dom}(\text{channel\_content}) = \text{channel\_state}^{-1}[\{\text{busy}\}] \\
& \text{channel\_state} \in \text{CHANNELS} \rightarrow \text{state} \\
& \text{dom}(\text{channel\_state}) = \text{ran}(\text{mk\_channel}) \\
& \text{ran}(\text{channel\_content}) \subseteq \text{dom}(\text{moving\_pool}) \\
& \forall \text{msg} \cdot \text{msg} \in \text{dom}(\text{moving\_pool}) \wedge \text{des}(\text{msg}) = \text{moving\_pool}(\text{msg}) \\
& \Rightarrow \text{msg} \notin \text{ran}(\text{channel\_content})
\end{aligned}$$

Figure 7.10: M_1 : Invariants (channels)

The *switching* event is now refined to transfer a message to the next router through channels. In order to model this, we add a new event *out_to_channel* to model pushing a message in the channel. This event is enabled when there is a message for transferring in a node and the channel between the node and the next node is free. In addition, we refine the *switching* event to model releasing the channel by receiving the message at the end of the channel. This event is enabled when a message is in the channel.

The Third Model M_2

In this model, we define buffers for the ports of the nodes and refine the second model to model the communication in 3D NoCs by considering these buffers.

The Static Part. The context of the third model contains a single constant $buffer\_size \in \mathcal{N}_1$, which is a strict natural number denoting the maximum number of messages allowed in a buffer.

The Dynamic Part. Each node has fourteen buffers, each assigned to node ports; those assigned to output ports are called *output buffers* and those assigned to input ports are called *input buffers*. When there is a message in an output buffer of a node, the node can transfer it to the channel provided that the channel is free. If in the other side of the channel the input buffer has an empty place, the message is transferred to the input buffer of the next node and the channel is released; otherwise, the channel will be busy until an empty place appears in the input buffer. To model the buffer structure in the third model we add a new machine variable *buffer_content* that models the current content of all buffers. Indeed, adding and removing messages in/from buffers is modelled by the *buffer_content* variable.

In order to guarantee the correctness of the buffer modelling, we need the invariants shown Fig.7.11. They model that the content of a buffer never becomes more than its size. In addition, while a message is in the *moving_pool*, i.e., it has not reached to its destination, it must be either in a channel or in a buffer.

The *switching* event is refined to be enabled when there is an input buffer with at least one empty place at the end of the channel. Then, besides releasing the channel, the message in the channel is transferred to the input buffer. The status of the *switching* event is still anticipated since we do not store. The *out_to_channel* event, as shown in Fig.??, is refined to be enabled when there is a message in an output buffer meaning that the message is

VARIABLES
<code>buffer_content</code>
INVARIANTS
$buffer\_content \in \text{MESSAGES functional } (ROUTER \times PORTS)$
$dom(buffer\_content) \cup ran(channel\_content) = dom(moving\_pool)$
$dom(buffer\_content) \cap ran(channel\_content) = \emptyset$
$\forall b \cdot b \in ran(buffer\_content) \Rightarrow card(buffer\_content \triangleright \{b\}) \in 1 \dots buffer\_size$

Figure 7.11: M_2 : Invariants (buffer)

removed from buffer. The *sent_message* and *received_message* events are refined so that they update the *buffer_content* variable.

Event <i>routing</i> $\hat{=}$
extends <i>routing</i>
any
<code>msg router in_p out_p</code>
where
$in\_p \in \{Win, Ein, Sin, Nin, Uin, Din, Lin\}$
$out\_p \in \{Wout, Eout, Sout, Nout, Uout, Dout, Lout\}$
$(in\_p = Win \wedge out\_p \neq Wout) \vee (in\_p = Ein \wedge out\_p \neq Eout)$
$\vee (in\_p = Sin \wedge out\_p \neq Sout) \vee (in\_p = Nin \wedge out\_p \neq Nout)$
$\vee (in\_p = Uin \wedge out\_p \neq Uout) \vee (in\_p = Din \wedge out\_p \neq Dout)$
$\vee (in\_p = Lin \wedge out\_p \neq Lout)$
$msg \mapsto (router \mapsto in\_p) \in buffer\_content$
$card(buffer\_content \triangleright \{router \mapsto out\_p\}) < buffer\_size$
then
$buffer\_content(msg) := router \mapsto out\_p$
end

Figure 7.12: M_2 : Routing Event

At this level of abstraction, we refine the *routing* event (modelled as *skip* in the previous models) as shown in Fig.7.12. A routing algorithm decides on choosing an output channel for a message in an input channel. As we present a general model, we do not consider any specific routing algorithm and we model routing decision nondeterministically. That is, when there is a message in an input buffer of a node, it can be routed to any output buffer of the node except the output buffer in the same direction with the input buffer e.g., a message in the northern input buffer cannot be routed to the northern output buffer. We also check that there is enough space in the chosen buffer. We have this constraint to prevent a cycling problem in the communication that would lead to deadlock in the interconnection network.

We do not change the status of the *switching* event in this refinement step. Thus, its status is still anticipated. In order to have it convergent,

we need to define a variant based on some ordering relation of the message identifiers. This can be achieved by modelling a channel dependency graph but is not considered here.

A more concrete 3D NoC design can be modelled by refining one or more of these three general models and by verifying whether the design can guarantee the overall communication correctness. In the following, we model the XYZ routing algorithm by refining the third model M_2 and verifying whether it guarantees the overall communication correctness.

7.1.3 Case Study: the XYZ Routing Algorithm

In this section, we formally develop a dimension-order routing (DOR) algorithm which is a deterministic routing scheme widely used for NoCs [MKMA09]. To make the best use of the regularity of the topology, the dimension-order routing transfers packets along minimal paths in the traversing of the low dimension first until no further move is needed in this dimension. Then, they go along the next dimension and so forth until they reach their destination. For example, the dimension-order routing in the 3D NoC called the *XYZ routing algorithm* uses Z dimension channels after using Y and X dimension channels. Packets travel along the X dimension, then along the Y dimension and finally along the Z dimension. Thus, if $current\_node = (c_x, c_y, c_z)$ is a node containing a message addressed to node $destination = (d_x, d_y, d_z)$, then the XYZ routing function $R_{xyz}(,)$ is defined as follows:

$$R_{xyz}((c_x, c_y, c_z), (d_x, d_y, d_z)) = \begin{cases} (c_{x-1}, c_y, c_z) & \text{iff } c_x > d_x \\ (c_{x+1}, c_y, c_z) & \text{iff } c_x < d_x \\ (c_x, c_{y-1}, c_z) & \text{iff } c_x = d_x \wedge c_y > d_y \\ (c_x, c_{y+1}, c_z) & \text{iff } c_x = d_x \wedge c_y < d_y \\ (c_x, c_y, c_{z-1}) & \text{iff } c_x = d_x \wedge c_y = d_y \wedge c_z > d_z \\ (c_x, c_y, c_{z+1}) & \text{iff } c_x = d_x \wedge c_y = d_y \wedge c_z < d_z \end{cases}$$

In order to model the XYZ routing algorithm based on the third general model, we have to refine the routing event which is nondeterministically defined. As shown in the above formula, a message can be transferred to six different directions based on its current position and destination. Therefore, we refine the *routing* event in the previous model to six *routing* events so that their guards are based on the routing formula. As an example of the *routing* event, we show in Fig.7.13 the situation where c_x is greater than d_x . All the correctness properties defined for the abstract models are proved. Hence, the XYZ routing algorithm guarantees the overall communication correctness.


```

Event   routing_X_dec  $\hat{=}$ 
extends routing
where
    coordX(router) > coordX(des(msg))  $\wedge$  out_p = Wout
then
    buffer_content(msg) := router  $\mapsto$  out_p
end

```

Figure 7.13: The XYZ Model: Routing Event ($c_x > d_x$)

7.1.4 Verification of the Models

In order to prove that the models satisfy their correctness properties we have to check that they respect their invariants, i.e., the pool properties for our models. To prove this, we have generated the proof obligations for all the models using the Rodin tool: part of the proof obligations were automatically discharged and the rest of could be proved interactively. The proof statistics for our models are shown in Table 1. These figures express the number of proof obligations generated by the Rodin platform as well as the number of obligations automatically discharged by the platform and those interactively proved. A high number of interactive proofs were due to reasoning about set comprehension and unions, not currently supported automatically in Rodin. In addition, the interactive proving often involved manually suggesting values to discharging various properties containing logical disjunctions or existential quantifiers. Extra proving was due to the fact that currently, we cannot create proof scripts and reuse them whenever needed in Rodin. Thus, in some cases we had to manually repeat very similar or almost identical proofs.

Table 7.1: Proof Statistics

Model	Number of Proof Obligations	Automatically Discharged	Interactively Discharged
Context	21	6(28%)	15(72%)
M_0 Model	38	34(89%)	4(11%)
M_1 Model	33	11(33%)	22(67%)
M_2 Model	33	7(21%)	26(79%)
XYZ Model	13	0(0%)	13(100%)
Total	144	64(45%)	80(55%)

7.1.5 Conclusions

In this report, we have proposed the abstract models M_0 , M_1 , and M_2 at three increasing levels of detail for 3D NoCs. These can be used for modelling and verifying 3D NoC-designs in the early stages of the system development. We have also shown how to apply such an abstract model to verify a concrete 3D NoC routing algorithm. Most importantly, the overall correctness of the communication models (expressed using a special data structure called pool [Abr10]) is guaranteed for the 3D NoCs. We have achieved this by modelling the correctness condition via invariants; as each model added detail to the previous model, the invariant needed to reflect these added details in a consistent manner. In order for the invariant to be satisfied by a model, a number of proof obligations needs to be discharged. Moreover, in order for the models to respect the refinement relation $M_0 \sqsubseteq M_1 \sqsubseteq M_2$, i.e., to develop each other in a provably correct manner, some other proof obligations need to be generated. As we have employed the Rodin platform to specify our 3D NoC modelling, many of these proof obligations have been automatically discharged, while for the rest it was possible to discharge them interactively. We note an interesting property of our communication correctness condition, that essentially reduces to the fact that all the messages will eventually reach their destinations. This is a typical liveness property that we model here as an invariant, also based on the variant expression ensuring that our models will eventually terminate. The liveness property can also be verified via a model checker, for instance Pro-B [Pro], that is associated to the Rodin platform.

7.2 Model-Based Analysis Tools for Component Synthesis

7.2.1 Introduction

Component-based development identifies and manages interdependencies among preexisting software or hardware parts for integrating them into new systems [Del97], [Szy98]. During the last two decades, it has been widely recognised that methodologies and frameworks with adequate tool support are needed, in order to facilitate the component-based development.

In this work, we propose a formal method based methodology for component synthesis. We understand the functionality of a component as a collection of *services* that the component has to implement [Szy98]. Components have been traditionally developed with an emphasis on the spec-

ification of their functionality, without explicitly describing and analysing their interdependencies. Here, based on the services that components need to implement, we identify some initial boundaries of components. Then, we identify those services needed for communication among components; hence, we identify the component interdependencies. Based on these interdependencies, we then reason about the suitability of certain boundaries, i.e., we argue whether some components should or not merge. As an application of our approach, we employ these interdependencies to place components on hardware platforms, so that highly communicating components find each other in their vicinity.

The synthesis approach is presented in the B Action Systems formalism in combination with the ProB tool. However, it can be also straightforwardly applied while using Event-B (the Rodin platform) and its ProB extension.

7.2.2 Preliminaries

In this section we present the formalisms we employ in this work together with an example modelled with these formalisms.

B Method Based Formalisms

B Action Systems [WS98] is a state-based formalism based on Action Systems [BKS83] and the B Method [Abr96]. This framework was developed in order to allow reasoning about parallel and distributed systems within the B Method.

We illustrate the form of a B Action System in Fig. 1 (a). The main computational unit is the *machine*, identified by a unique name A. A machine has a finite set of (local) variables and a finite set of operations that evaluate and modify the variables. The variables of the machine are declared in the **VARIABLES**-clause; their values describe the *state* of the machine. The **INVARIANT**-clause defines the types of the variables and gives their guaranteed behaviour. Initial values are assigned to the variables in the **INITIALISATION**-clause. The operations in the **OPERATIONS**-clause are of the form $\text{Oper} = \mathbf{SELECT\ P\ THEN\ S\ END}$, where P is a predicate (called *guard*) on the variables and S is a substitution (update) statement; hence, an operation can evaluate and modify the state of the system modelled by the variable values. When P holds, the operation Oper is said to be enabled. Only enabled operations are considered for execution and if there are several operations enabled simultaneously then only one is selected for execution in a non-deterministic manner. If some operations have no variables in common and are enabled at the same time, then they can be considered to execute in

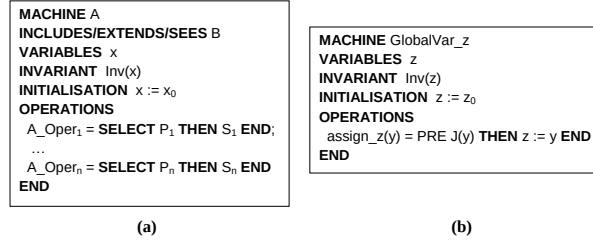


Figure 7.14: (a) Example of a B Action System, (b) Global variables in B Action Systems

parallel since their sequential execution in any order gives the same result. When there are no enabled operations the machine terminates.

Structuring mechanisms can be used to express B action systems as a composition of subsidiary systems. The mechanisms used in this work are the **SEES** and **INCLUDES** mechanisms [Abr96]. The **SEES**-mechanism allows reading access to the seeing machine, meaning that variables of the seen machine can be used in the initialization and operations of the seeing machine. When using the **INCLUDES**-mechanism, in addition to the reading access, the invariant of the including machine can express requirements on the variables of the included machine. The variables of the included machine are directly visible to the including machine, but they may only be updated via the included system.

Global variables that can be read and updated by more than one system are one of the most common communication mechanisms in B Action Systems. A global variable z is declared in a separate machine as shown in Fig. 1 (b). This machine is then **INCLUDED** or **SEEN** in the machines that refer to the global variable and the including machine is allowed to assign a new value y to the variable z via an operation `assign_z(y)` [WS98] in the global variable machine. Global variables are further employed in the next section.

Parallel composition of several machines into a single machine is an important aspect of our modelling approach as will become apparent in the following sections. The parallel composition of a machine A and a machine B is illustrated in Fig. 2 and it is formed by merging the variables, invariants and operations of A and B. The local variables of the machines have to be distinct. This can easily be achieved by renaming the conflicting variables before forming the composition. The global variables declared in a global variable machine will be the global variables of the parallel composition. Since the invariant of the composed machine is the conjunction of the invariants of the individual machines before the composition, the operations of each machine should preserve the invariants of all other machines in the composition. This

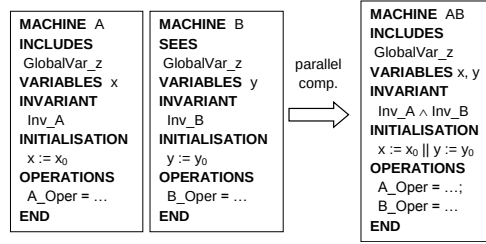


Figure 7.15: Parallel composition of B Action Systems A and B

is mainly a restriction on the assignments to the common global variables.

ProB tool. In order to analyse B Action System specifications we employ the ProB tool [LB03]. ProB is a model checking and animation tool for B machines that includes a fully automatic animator written in SICStus prolog [SIC]. ProB takes an instantiated model in B, i.e. a model in which any generic set has been instantiated with some concrete values to avoid state explosion and generates a finite coverage graph. In this work we only employ the animation capabilities provided by ProB.

ProB also computes the *coverage* of the model after a random animation of a number of operation executions. The computed coverage consists of information about the total number of the nodes of the state space of the system and their status (*deadlocked*, *live*, or *open*). It also provides the total number of operations of the model, as well as the possibly covered operations between the nodes. The list of **DEFINITELY_ENABLED_AFTER** operations and the list of **POSSIBLY_ENABLES** operations are also provided. The former list contains the operations that are definitely enabled after the execution of a subset of the model's operations. The latter list contains the operations that are possibly enabled after the execution of each operation of the model. Furthermore, the *Signature-Merge Reduced Statespace* (S-MRS) graph contains specific transitions (operation executions) based on some specific global state, for instance, we can observe the different starting points of possible interleaving. This S-MRS graph corresponds to the full state-space graph.

GeneSyst tool. S-MRS graphs from ProB are prone to rapid expansion if the model is non-deterministic and the sets defined for variable declarations are large. Such graphs can then be difficult to examine and of small value to the user of the tool. A potential alleviation of this problem comes from animating tools like GeneSyst [BP05]. This is an animating tool for B machines that takes as input a set of disjoint predicates (given by the user in an additional **ASSERTIONS**-clause) on the state variables of a B machine.

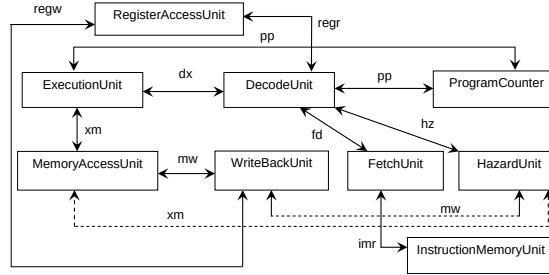


Figure 7.16: Asynchronously communicating subsystems of a pipelined processor

Based on several internal checks and proofs, the tool outputs an abstract state-transition diagram corresponding to the predicates given by the user.

An Example Modelled with the Formalisms

As a case study, we consider a model of an asynchronous pipelined processor introduced by Plosila and Sere [PS97] and specified originally within the Action Systems formalism [BKS83]. In the following we describe this model to the extent needed in this work and put forward the B Action Systems and ProB specifications of this model.

In Fig. 3 we illustrate several components of the processor together with their dependencies: FetchUnit, InstructionMemoryUnit, DecodeUnit, RegisterAccessUnit, ProgramCounter, HazardUnit, ExecutionUnit, MemoryAccessUnit and WriteBackUnit. The components cooperate in the following manner. First, FetchUnit fetches a new instruction for manipulation from InstructionMemoryUnit and initiates the communication with DecodeUnit. DecodeUnit then communicates with HazardUnit in order to avoid a *pipeline hazard*, i.e., a read-write conflict on the relevant registers between a new incoming instruction in DecodeUnit and a current instruction being written to the memory. When the memory is updated with the current instruction, HazardUnit acknowledges back and allows DecodeUnit to increment ProgramCounter, read the register, and decode the new instruction. DecodeUnit waits for the acknowledgements from ProgramCounter and RegisterAccessUnit before it acknowledges the communication with FetchUnit and sends the current decoded instruction to ExecutionUnit. After the communication with FetchUnit is acknowledged, a new instruction can be fetched. ExecutionUnit manipulates the instruction: depending on its type, it increments or loads ProgramCounter. After the acknowledgement from ProgramCounter, ExecutionUnit initiates communication with MemoryAccessUnit and acknowledges the communication with DecodeUnit, so that it can receive a new decoded instruction. MemoryAccessUnit sends the instruction to WriteBackUnit, that writes the

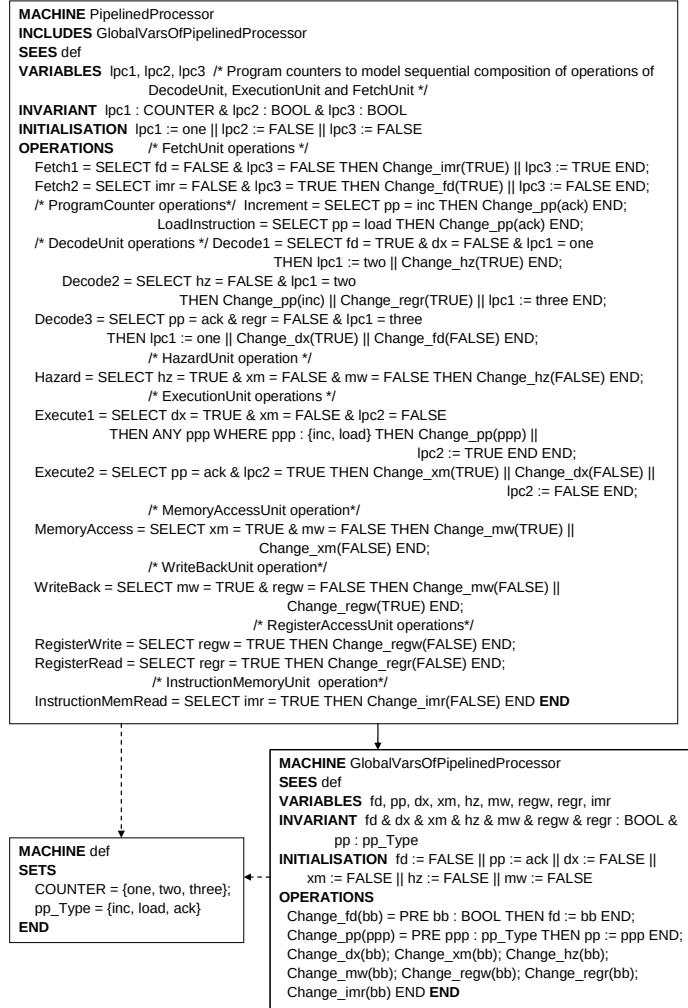


Figure 7.17: B Action System of PipelinedProcessor

instruction to the memory through RegisterAccessUnit and acknowledges back to MemoryAccessUnit the memory update. The manipulation of the next instruction can then continue from an intermediate pipelined stage. More details of the model appear in [PS97].

The dependencies between the components are modelled with global variables. The dashed arrows indicate read access to the attached variables *mw* and *xm* in HazardUnit. The continuous line arrows indicate read and write access to the attached variables. For example, the components ProgramCounter,

DecodeUnit and ExecutionUnit are all allowed to update the global communication variable `pp`, that can be updated with the values `inc` (to increment ProgramCounter), `load` (to load ProgramCounter), and `ack` (acknowledgement from ProgramCounter for the two possible updates). The other global variables are of type `BOOL` to model the request and acknowledgement phases of the asynchronous communication. The value `TRUE` corresponds to the *request* phase and the value `FALSE` corresponds to the *acknowledgement* phase.

For applying the analysis techniques provided by ProB, the decomposed model of the processor illustrated in Fig. 3 is composed (in parallel) into a single B Action System as defined in Section 7.2.2. Currently, ProB can model check and animate only one system and not a collection of systems running in parallel at the same hierarchical level. However, this is not a real restriction since the execution of various systems in parallel is equivalent to their parallel composition as explained in Section 7.2.2.

In Fig. 4 we illustrate the specification of the pipelined processor. The shared global variables together with simple operations to update them are defined in the separate global variable machine `GlobalVarsOfPipelinedProcessor`. The required sets for this specification are defined in a separate machine `def`. The continuous line arrow indicates the **INCLUDES** structuring mechanism and the dashed line arrow indicates the **SEES** mechanism. `PipelinedProcessor` has three (local) variables named `lpc1`, `lpc2` and `lpc3` to model the sequential composition of the operations of `DecodeUnit`, `ExecutionUnit` and `FetchUnit`, respectively. The operations of each component are grouped together and the components are conventionally separated from each other with a comment line. For instance, the operations between the comment lines `/* FetchUnit operations */` and `/* ProgramCounter operations */` denote the `FetchUnit`, the operations between the comment lines `/* ProgramCounter operations */` and `/* DecodeUnit operations */` denote the `ProgramCounter` and so on. The functionality of the subsystems in Fig. 4 corresponds to the description of their dependencies in Fig. 3.

We observe here the nature of this case study. In the model illustrated in Fig. 3, the variables are stored within the various components and declared global, so that other components can also access them. The control flow in the pipelined processor is mainly sequential and then generates various branches of parallel flows. One component can start its job only if specific actions have taken place before that. We translate this component sequencing to communication modeled in B Action Systems via the **SEES** and **INCLUDES** mechanisms.

We now observe the nature of the B Action Systems model of `PipelinedProcessor`. The most obvious difference with respect to the model illustrated in Fig. 3 is that the global variables are no longer stored within the units but

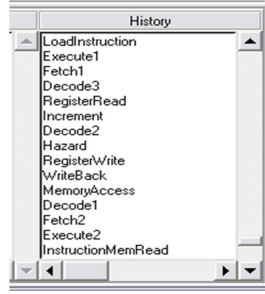


Figure 7.18: Implementation of PipelinedProcessor in ProB and history of executed operations

instead in one **INCLUDED** machine. This is only a technical detail for specifying PipelinedProcessor in ProB. When we analyse the pipelined application, we assume the variables stored within the components as in the original model [PS97] illustrated in Fig. 3. In Fig. 5, we show the history of the executed operations after an animation; this is only a part of the main interface of ProB, where the model in Fig. 4 is specified. We focus on this part of the window because it is the most relevant for our component mapping method. The other parts of the main interface contain the specification under animation as well as the enabled operations and the state properties during animations.

7.2.3 The Synthesis Method

Our approach to synthesising components is based on the component services: they are the basic units of composition. We model component services with machine operations in the B Action Systems model; hence, we start our approach based on a B Action Systems model comprising all the possible operations to synthesise. Every specification typically comes with a semantical grouping of operations into components. We take advantage of this initial grouping; the purpose of our method is to determine more suitable component configurations if possible. We assume we are given a positive integer n that represents the maximum desired number of interdependencies among components. hence, the inputs of our method are a suggested delineation of components and the number n .

The B Action Systems model is translated into a ProB model on which we apply various animations techniques in order to challenge the component boundaries. Our proposed method is described below.

1. We run a (large enough) animation in ProB, based on the ProB model of the application. Based on the computed coverage of the ProB

model, we record the relations between the operations in the lists `DEFINITELY_ENABLED_AFTER` and `POSSIBLY_ENABLES`.

2. We replace each operation in the recorded lists of Step 1 with the name of its component.
3. We count the number of occurrences of the name of each component in the recorder lists created in Step 2. We store this information in a table with three columns: one for the component names, one for the number of communications of each component, modelled by the enabledness relations recorded in Steps 1-2, and one for the names of the communication partners of each component. If the same relation occurs in both lists `DEFINITELY_ENABLED_AFTER` and `POSSIBLY_ENABLES`, we only count it once. We place the component names in the table in the descending order of the number of their communications.
4. We check the table created in the previous step to identify components that communicate more than n times with other components, by observing the number of communications associated with component names in the table.
 - (a) If there is no such component, then we have a suitable component configuration and the method ends.
 - (b) If there are such components, then they have a too narrow border. We enlarge the border of the highest communicating component to comprise all the services of all its communication partners. We feed back the new component configuration to our method and restart from Step 1.

We run a large enough animation so that it produces a full state-space graph in ProB. By analysing applications in the above way we have a structured method for analysing the number of component interdependencies. Based on it we can change the component configuration to a desired level of component interdependencies.

Applying the Synthesis Method

As a case study of our method, we study the component interdependencies of the pipelined application described in Section 7.2.2. We apply our synthesis method starting with the ProB specification, where the components are delineated by comments as shown in Fig. 4. We assume that $n = 4$. When

DEFINITELY_ENABLED_AFTER Hazard==>Decode2 Decode2==>Increment Decode2==>RegisterRead Decode3==>Fetch1 Decode3==>Execute1 Execute2==>MemoryAccess MemoryAccess==>WriteBack Fetch1==>InstructionMemRead WriteBack==>RegisterWrite InstructionMemRead==>Fetch2 LoadInstruction==>Execute2 POSSIBLY_ENABLES Hazard?=>Decode2 Decode2?=>Increment Decode2?=>RegisterRead RegisterRead?=>Decode3 Decode3?=>Fetch1 Decode3?=>Execute1 Decode1?=>Hazard Execute2?=>MemoryAccess Execute2?=>Decode1 Increment?=>Execute2 Increment?=>Decode3 MemoryAccess?=>WriteBack Fetch1?=>InstructionMemRead WriteBack?=>RegisterWrite InstructionMemRead?=>Fetch2 RegisterWrite?=>Hazard Fetch2?=>Decode1 LoadInstruction?=>Execute2 Execute1?=>Increment Execute1?=>LoadInstruction	DEFINITELY_ENABLED_AFTER HazardUnit ==> DecodeUnit DecodeUnit ==> ProgramCounter DecodeUnit ==> RegisterAccessUnit DecodeUnit ==> FetchUnit DecodeUnit ==> ExecutionUnit ExecutionUnit ==> MemoryAccessUnit MemoryAccessUnit ==> WriteBackUnit FetchUnit ==> InstructionMemoryUnit WriteBackUnit ==> RegisterAccessUnit InstructionMemoryUnit ==> FetchUnit ProgramCounter ==> ExecutionUnit POSSIBLY_ENABLES HazardUnit ?=> DecodeUnit DecodeUnit ?=> ProgramCounter DecodeUnit ?=> RegisterAccessUnit RegisterAccessUnit ?=> DecodeUnit DecodeUnit ?=> FetchUnit DecodeUnit ?=> ExecutionUnit DecodeUnit ?=> HazardUnit ExecutionUnit ?=> MemoryAccessUnit ExecutionUnit ?=> DecodeUnit ProgramCounter ?=> ExecutionUnit ProgramCounter ?=> DecodeUnit MemoryAccessUnit ?=> WriteBackUnit FetchUnit ?=> InstructionMemoryUnit WriteBackUnit ?=> RegisterAccessUnit InstructionMemoryUnit ?=> FetchUnit RegisterAccessUnit ?=> HazardUnit FetchUnit ?=> DecodeUnit ProgramCounter ?=> ExecutionUnit ExecutionUnit ?=> ProgramCounter ExecutionUnit ?=> ProgramCounter

Figure 7.19: Part of the computed coverage of PipelinedProcessor

applying Step 1 in our method we get the lists described in the left part of Fig. 6.

After applying Steps 2 and 3, we obtain the table shown in Fig. 7. This table shows for each component the number of communications with other components as well as the names of these components. The definite and possible dependencies between the components are shown in the right part of Fig. 6. Studying the table we can apply Step 4 of our method, where we reach the conclusion that DecodeUnit should be merged with FetchUnit, RegisterAccessUnit, ProgramCounter, HazardUnit, and ExecutionUnit in order to have more suitable component boundaries.

The method should now be applied again on the new component configuration. We do not show it here due to lack of space, but one can easily see that in the new configuration there will only be four components, with at most four communications each. Hence, the method now stops.

On the Correctness of our Method

The method takes as input a given component configuration $\mathcal{C} = \{C_1, \dots, C_m\}$ so that each component C_i , $i \in \{1, \dots, m\}$ implements a certain number p_i of services S_j , $j \in \{1, \dots, p_i\}$: $C_i = \{S_j | j \in \{1, \dots, p_i\}\}$. Some of these services may denote internal computation, while others may enable some

Components	No of Component Interactions	Communication Partners
DecodeUnit	10	FetchUnit, RegisterAccessUnit, ProgramCounter, ExecutionUnit, HazardUnit
ExecutionUnit	5	DecodeUnit, MemoryAccessUnit, ProgramCounter
FetchUnit	4	DecodeUnit, InstructionMemoryUnit,
RegisterAccessUnit	4	DecodeUnit, WriteBackUnit, HazardUnit
ProgramCounter	3	DecodeUnit, ExecutionUnit
HazardUnit	3	DecodeUnit, RegisterAccessUnit
MemoryAccessUnit	2	ExecutionUnit, WriteBackUnit
InstructionMemoryUnit	2	FetchUnit
WriteBackUnit	2	MemoryAccessUnit, RegisterAccessUnit

Figure 7.20: Dependencies between the components of PipelinedProcessor, their number of interactions and their communication partners

of the services of other components. The latter occurrence is detected by our method via the ProB animation tools and we refer to it as *component communication*. We identify with our method a certain number r_i for each component $C_i, i \in \{1, \dots, m\}$ that denotes the number of definite and possible communications for the respective component $C_i, i \in \{1, \dots, m\}$. We denote with max_k the highest number of component communications at run k of the method, for instance $max_1 = 10$ and $max_2 = 4$ in our example above. We also take as input to the method a positive integer n , denoting the maximum desired number of component interdependencies. The goal of the method is to determine a component configuration $\mathcal{C}' = \{C'_1, \dots, C_{m'}\}$ where each $r'_i \leq n, i \in \{1, \dots, m'\}$.

Termination. As we start with a finite number m of components and each rerun of the method is triggered by some of these components merging, we observe that $m < m'$. This necessarily means that the method terminates, in the extreme case with $m' = 1$ and $r'_i = 0$. The extreme case thus translates to all the components merged into one.

From one configuration to the next. While we are certain that the method terminates with $0 \leq r'_i \leq n, i \in \{1, \dots, m'\}$, we cannot guarantee that the highest number max_k of component communications at each run k of the method is continuously decreasing. For instance, Fig. 8 shows a configuration $\mathcal{C} = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ where $r_1 = 20, r_2 = 18, r_3 = 18, r_4 = 16, r_5 = 12$, and $r_6 = 12$ and $n = 12$. We also assume that C_1 communicates 10 times with C_2 , 6 times with C_3 , and 4 times with C_4 ; C_2 communicates 10 times with C_1 , 4 times with C_3 , and 4 times with C_5 ; C_3

Components	No of Component Interactions	Communication Partners
C ₁	20	C ₂ , C ₃ , C ₄
C ₂	18	C ₁ , C ₃ , C ₅
C ₃	18	C ₁ , C ₂ , C ₆
C ₄	16	C ₁ , C ₅ , C ₆
C ₅	12	C ₂ , C ₄
C ₆	12	C ₃ , C ₄

Figure 7.21: Dependencies and number of interactions between components $C_1, C_2, C_3, C_4, C_5, C_6$

communicates 6 times with C_1 , 4 times with C_2 , and 8 times with C_6 ; C_4 communicates 4 times with C_1 , 8 times with C_5 , and 4 times with C_6 ; C_5 communicates 4 times with C_2 and 8 times with C_4 ; and C_6 communicates 8 times with C_3 and 4 times with C_4 . This means that after the first run of the method, the first four components are merged; we have $max_1 = 20$ while $max_2 = 24$.

This is not a wrong result as the following run of the method will provide a configuration where all the components are merged into one and the method ends. However, this type of result shows that the component configuration contains a high number of interdependencies that are best avoided if all the components are merged. Our method is intended for analysis purposes and by running it several times we may decide that perhaps the initial configuration with $max_1 = 20$ was more balanced than the second one with $max_2 = 24$. Obviously, the method is highly dependent on the maximum desired number of component interdependencies, n .

The desired number of component interdependencies. The desired number of component interdependencies is relative to each application. In some cases, it can depend on some standards and in some cases, it can depend on the experience of the developers. One can also start from an arbitrary number n and check the generated component configuration, given the initial configuration and such n . If the configuration is not suitable with the purposes of the application, then a higher interdependency level might be tested.

The ProB animations are instrumental in our method as they allow us to determine significant numbers related to component communication. One can see from figures such as Fig. 3 that DecodeUnit communicates with five other components; however, it is not clear how much communication actually occurs on these five ‘channels’.

Another interesting application of our method refers to placing components on hardware platforms so that the highly communicating components are placed close to each other. We study this particular problem in more detail in the next section.

7.2.4 Applying the Synthesis Method to NoC Mapping

Recently the Network-on-Chip (NoC) communication paradigm [HJK⁺00] has been proposed as the intercommunication scheme of the cores in Multi-Processor Systems-on-Chip (MPSoC) offering sufficient bandwidth for concurrent on-chip data transactions, incremental scalability of the network, as well as distributed and flexible routing of data.

The mapping of the components of an application to NoC platforms and their subsequent communication patterns contribute essentially to the efficient execution of the application as well as to the performance and power consumption of the NoC system. Meeting these *efficiency* constraints for various design requirements becomes a challenge due to the growing complexity of systems [MOP⁺09]. To address this, application mapping techniques have been recently proposed [HM03] with the main advantage of reducing the length of the global interconnect which in turn reduces power consumption considerably. In other words, the main contribution of application mapping techniques is to place communicating components as close as possible to each other such that the energy required to transfer data between them is minimised. Hence, the mapping of an application to a NoC platform considerably influences the efficiency of a NoC regardless of the nature of the communication patterns.

We base our synthesis method for efficient application mapping on several assumptions. First, we assume that some NoC communication topology is available for the actual propagation of data between the components of an application. Second, we assume that a certain application can be first generically developed and then "applied" to various network settings. Hence, it is not necessary to consider the network mechanisms when designing the application, but rather have them integrated at a later stage. This separation of concerns is important for reusing the generic application and port it on various platforms. We also assume that neighbouring tiles on a NoC communicate faster than non-neighbouring tiles, hence our goal is to place components that communicate heavily as close as possible to each other.

Based on these assumptions, given an already specified application, we explore the mapping of the application's software components to NoC tiles with the ProB and GeneSyst formal approach to handle and analyse the dependencies between the components. A simplified 2D mesh NoC topology

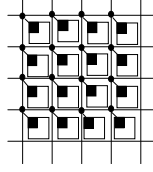


Figure 7.22: A 2D NoC mesh

is illustrated in Fig. 9. The dots represent the NoC routers and the white squares represent processing cores as well as storage elements connected to their routers through network interfaces. The interfaces are represented by the black squares within the white squares.

We adapt our synthesis method described in Section 7.2.3 to mapping components onto a NoC platform. In Step 4 in our method, instead of defining the new component, we simply place the communicating components as close as possible to the highly communicating one. Moreover, we incorporate the additional ProB facilities to help understand the pipelined traces as intermediate steps in our method, as well as we employ the GeneSyst tool for facilitating the placement in the last step. The steps of our generic method of Section 7.2.3 are therefore modified to:

1. We run a (large enough) animations in ProB based on the ProB model of the application.
2. We generate an abstract state-transition diagram with GeneSyst based on specific states of the application.
3. Based on the computed coverage of the model by ProB, we record the relations between the operations of the model's components given in the lists `DEFINITELY_ENABLED_AFTER` and `POSSIBLY_ENABLES`.
4. We enrich the operation traces with information about interleaving operations by:
 - (a) observing the history window of ProB for the generated operation traces.
 - (b) observing the computed coverage and the S-MRS graph of the model to understand the different starting points of the interleaving.
5. We replace each operation in the recorded lists of Step 3 with the name of its component.

6. We count the number of occurrences of the name of each component in the table created in Step 5 and store this information in another table. If the same relation occurs in both lists `DEFINITELY_ENABLED_AFTER` and `POSSIBLY_ENABLES`, count it only once. We place the components in the table in descending order of the number of their occurrences.
7. We identify the component that has communication with the biggest number of other components by observing the number associated with it in the previous step and place that component in the 2D NoC mesh at a position where there are enough free tiles around it for the placement of the other components.
8. We parse the rest of the table created in Step 6 and place each remaining unplaced component on the NoC according to the relationships with the other units stored in the table created in Step 6, the interactions shown in the enriched trace created in Step 4 and the information observable by the abstract state-transition diagram generated by GeneSyst in Step 2.
 - (a) If there are components interfering only with one other component, we place them beside the interfering component and out of the way of communication between the other components.
 - (b) We perform the placing of the rest of the components according to the interactions of their operations in relation to the operations of the placed components.

We exemplify our mapping method based on the specification of `PipelinedProcessor` in ProB. After application of all the steps of our method we propose the mapping of the components of `PipelinedProcessor` as shown in Fig. 10. We note that any NoC communication topology can be considered, but in this work we assumed a 2D mesh as the target topology. The proposed mapping is justified as follows.

We first identify `DecodeUnit` as the component that has communication with the biggest number of other components (Step 7). Thus, it is placed on the 2D NoC mesh at a position where there are enough free tiles around it for the placement of the other components. We parse repeatedly the rest of the table (Step 8) and place `ExecutionUnit` beside `DecodeUnit` with a distance between them of one hop, i.e. one connecting line between their corresponding routers shown in Fig. 10. The reason for this placement is because these components communicate regularly according to the dependencies and pipelining observable after the application of our method. Moreover, they

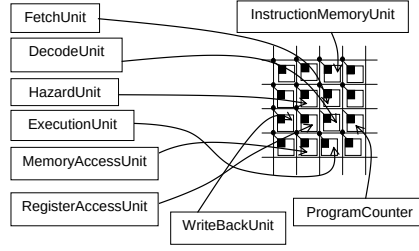


Figure 7.23: Placement of the subsystems of PipelinedProcessor on a region of a NoC 2D mesh

both communicate with several non-communicating components that do not need to be placed close together. The third most communicating component is RegisterAccessUnit and is placed beside DecodeUnit because they communicate. RegisterAccessUnit does not interact with FetchUnit, thus they do not need to be placed close to each other.

We continue with the placement of ProgramCounter. It interferes only with DecodeUnit and ExecutionUnit which results in placing these three components close together and ProgramCounter to be out of the way of the communication between DecodeUnit, ExecutionUnit and the rest of the components. In a similar manner we can decide on the placement of HazardUnit. Based on the previous analysis it communicates with DecodeUnit and always allows DecodeUnit to continue computation after operation WriteBack of WriteBackUnit and the update of RegisterAccessUnit. MemoryAccessUnit communicates with ExecutionUnit and WriteBackUnit, which in turn communicates with HazardUnit. Thus, MemoryAccessUnit, ExecutionUnit, WriteBackUnit and HazardUnit need to be placed close together. InstructionMemoryUnit interacts only with FetchUnit, so it is placed beside FetchUnit without interfering with the other units. Considering all these interactions which are based on the analysis in Section 7.2.4, we decide on the final placement shown in Fig. 10.

We can justify the proposed placement of the components of PipelinedProcessor based on one of the main metrics for efficient application mapping in the literature, namely *total communication cost*. This metric is calculated by summing all the products of required bandwidth and the distance in hops on the NoC for each pair-wise communication. If we assume that the required bandwidth for each pair-wise communication is equal then the decided placement incurs the minimum total communication cost for the execution of the application. If the bandwidth required for each pair-wise communication was known then this would form one more input to our method. This will be considered in future applications of our method.

7.2.5 Conclusions

Formal methods with adequate tool support are important for the design and analysis of complex systems in order to correct errors in the early design phases and reduce the involved costs of system design and development.

In this work we propose a novel method for analysing component boundaries, based on a desired interdependency level between component services. The component interdependencies are determined based on ProB animation facilities such as the definite and the possible communications for one component. This is a decision we assume for our method here, but elsewhere [Pet11] we have devised a method based only on the definite communications. We demonstrate the suitability of our proposed method to efficient application mapping to a NoC platform, which we believe is one potential application domain of our method.

To the best of our knowledge, this is the first approach to use model checking and animating tools of formal methods in order to facilitate the analysis of component interdependencies. This development is thus important in order to design dependable systems.

Specific to applying our analysis method for facilitating efficient application mappings to NoC platforms, in this work we did not consider the actual computation volume of each subcomponent, nor the required bandwidth for the component intercommunication which affects the total communication cost for the mapped application. These are additional metrics for deciding the mapping of an application to NoC platforms and will be considered in future applications of our method.

Chapter 8

Concurrency

J.R. Abrial, T. Hoang, C. Jones

8.1 Concurrency

We consider here programs that use several co-operating parallel processes in order to compute the intended final result. Proving correctness of such programs is a difficult task because of the interleaved execution of many sub-statements from different processes. These sub-statements may be executed in an unpredictable order. As a result, techniques such as program testing do not give us sufficient confidence about the correctness of these programs, since no execution leading to an error might appear during tests. To achieve correctness, it is therefore necessary to develop these programs and prove them formally.

There are a number of methods for proving the correctness of parallel programs [dRdBH⁺01]. Our main contribution is an approach applying the technique of refinement and decomposition in Event-B [Abr09]. The approach contains four steps as follows.

Stage 1 Starts with an abstract specification *in-one-shot* giving the purpose of the program.

Stage 2 Refines this abstract specification by introducing details about the *shared variables*.

Stage 3 Decomposes the model in the previous step to split the model into several (abstract) sub-models for processes.

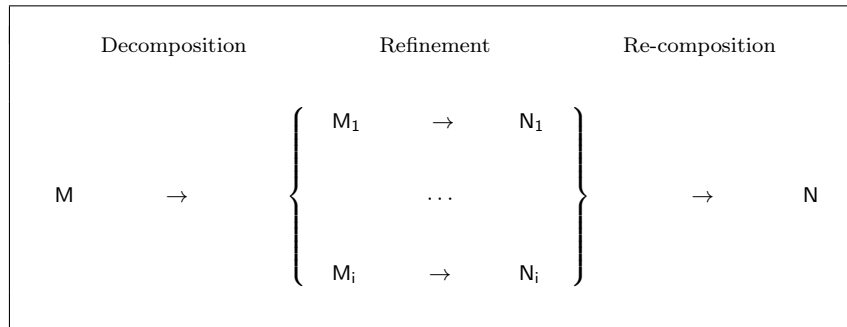
Stage 4 Refines each sub-model in the previous step independently.

In the last step, each sub-model can be seen as a new abstract specification, hence application of **Stage 2**, **Stage 3** and **Stage 4** can be repeated again. The novelty of our approach is in **Stage 2** where we specify shared information between processes. This information has dual purpose. Firstly, it contains the necessary guarantee condition from each process to establish the final result. Secondly, it also gives the condition for which each process can rely on in further development. This decision, i.e., to have this step early in our development, takes advantage of decomposition technique and results in simpler models and reduces the complexity of proving programs. This is the main advantage of our method over existing approaches.

Section 8.1.1 gives an overview of shared-variable decomposition in Event-B. Section 8.1.2 introduces the *FindP* program and its formal development using our approach is presented in Section 8.1.3. Section 8.1.4 compares our approach with some existing methods for developing parallel programs and we draw some conclusions in Section 8.1.5.

8.1.1 Shared Variable Decomposition

The idea of decomposition is to split a large model into smaller sub-models which can be handled more comfortably than the whole: one should be able to refine these sub-models independently. More precisely, if one starts from an initial (large) model, say M , decomposition allows us to separate this model into several sub-models $M_1 \cdots M_i$. These sub-models can then be refined independently yielding $N_1 \cdots N_i$. The correctness of the decomposition technique guarantees that the model N , obtained by re-composing $N_1 \cdots N_i$, is a refinement of the original model M . This process is illustrated in the following diagram:



Generation of sub-models using shared variable decomposition

Given a certain model M with events $e_1(a)$, $e_2(a, c)$, $e_3(b, c)$, $e_4(b)$,¹ we would like to decompose M into two separate models: M_1 dealing with events e_1 and e_2 ; and M_2 dealing with events e_3 and e_4 .

By giving the above *event partition*, we must also perform a certain *variable distribution*. This distribution can be derived directly from the information about the partitioning of events and the set of variables that they access. In our example, M_1 must have variables a and c , while M_2 must have variables b and c . As a result c becomes a *shared variable* between the two models which *cannot be data-refined*. In contrast, variables a and b are private variables of M_1 and M_2 and can be data-refined by their corresponding sub-refinements.

Moreover, in each sub-model, we need to have a number of *external events* to simulate how shared variables are handled in the non-decomposed model. These events are abstract versions of the corresponding internal events and use only the shared variables. In our example, M_1 will have an external event corresponding to e_3 (beside the internal events e_1 and e_2). Symmetrically, M_2 will have an external event corresponding to e_2 . Similar to shared variables, *external events* cannot be further refined. The summary about our sub-models is as follows.

Model	Private variables	Shared variables	Int. events	Ext. events
M_1	a	c	$e_1(a)$, $e_2(a, c)$	$(\text{ext\_})e_3(c)$
M_2	b	c	$e_3(b, c)$, $e_4(b)$	$(\text{ext\_})e_2(c)$

We also present a practical construction of the external event given its original event. This is illustrated below for an external event $(\text{ext\_})e_2$ in sub-model M_2 . Intuitively, this event is the *projection* of the original event, i.e., e_2 , on the state of the sub-model M_2 .

e_2	$(\text{ext\_})e_2$
any t where	any t, a where
$G(t, a, c)$	$G(t, a, c)$
then	then
$a, c : Q(t, a, c, a', c')$	$c : \exists a' \cdot Q(t, a, c, a', c')$
end	end

More detail on shared variable decomposition in Event-B can be found in [Abr09].

¹Note that the variables appeared in brackets denote those that are *accessed* by these events, e.g. appearing in guard or action of the corresponding event.

8.1.2 Example: FindP Program

Our running example is a standard problem in the literature for parallel programs. The purpose of the *FindP* program is to find the first index k of an array *ARRAY*, if there is one, satisfies some property P . Otherwise, if this index does not exist, i.e., none of the array elements satisfy P , the program returns $M + 1$, where M is the size of the array.

We are interested in the solution using two parallel processes to independently investigate the array which was given by Rosen [Ros76]. The processes in the original program works on the sets of even and odd indices separately. We present here a slightly generalised version of it where the two processes work on any two different parts of the array, denoted as *PART1* and *PART2*, which cover the entire domain of the array, but not necessarily disjoint.

The main idea of each process is to independently evaluate the value of the array in ascending order and to publish the first value that it finds. Moreover, from time to time, a process looks at the value that is published by the other process in order to know if it needs to continue the search or if it can terminate early.

The pseudo-code for the main program is given below. Here *index1*, *index2* are the two local indices, and *publish1*, *publish2* are the published results of the processes. In the end, when both processes terminate, the result taken is the minimum of the two published results.

```

index1, index2 := min(PART1), min(PART2);
publish1, publish2 := M + 1, M + 1;
process1 || process2;
result := min({publish1, publish2})

```

The pseudo-code for each process (presented here *process*₁) is as follows. Each process needs to continue only if its local index is smaller than both published results (as indicated by the guard of the loop). If this is the case, the process evaluates the value of the array at the current index and performs appropriate actions: publishing the current index or moving to the next index if possible.

```

while index1 < min({publish1, publish2}) do
  if ARRAY(index1) =  $\top$  then publish1 := index1
  else index1 := the-next-index-in-PART1-or-M+1 end
end

```

The key interaction between the two processes appears in the guard of the loop. Here the guard of *process*₁ refers to the published result of *process*₂, which in the meantime could be modified. In other words, *process*₁ needs to

read the published value of $process_2$ into some local variable before making the comparison using this local variable. The unfolded version of the $process_1$ is as follows. Our formal development in later sections is guided towards this version of the processes.

```

1 : (read)      read1 := publish2;
2 :             if index1 < min({publish1, read1}) then
                  if ARRAY(index1) = T then
(found)          publish1 := index1; goto 3;
                  else
(inc)            index1 := the-next-index-in-PART1-or-M+1; goto 1;
                  end
                  else
(not_found)      goto 3
                  end
3 : (end)

```

Here we make some assumptions on the atomicity. They are similar to the atomicity assumptions made by Abrial/Cansell [AC05].

- We have a number of shared variables (e.g. the published values). They are the variables that are written by one process and read by the other process. They are the shared variable with respect to the *read* process.
- We have a number of local variables (e.g. the local indices),
- The events involving only local tests and actions can be performed concurrently.
- There is an elementary atomic action for reading the value of a shared variable into a local variables, e.g. $local\_variable := shared\_variable$
- We extend the above atomic action to contain possible local test and local action.

```

when local_test then
    local_variable := shared_variable
    local_action
end

```

Different atomicity assumptions will lead to different *unfolded* versions of our program here. But this will not effect the applicability of our approach.

8.1.3 Formal Development

The machine-checked version of the development can be found on the web [Hoa09]. We first present our strategy for developing this program as follows.

Initial model specifies the result of the algorithm directly.

First refinement introduces the local indices of processes.

Decomposition step splits the model into sub-models corresponding to different processes: *main*, *process₁*, *process₂*.

We continue with further refinement steps for *process₁*; *process₂* should be developed in symmetrical fashion. Further development of the *main* process is straightforward and is not of our interest here.

First sub-refinement introduces the local index of the process.

Second sub-refinement introduces the read value of the process.

Third sub-refinement introduces the address counter for scheduling of events.

Initial Context and Model

The context defines an array of Booleans representing our abstract view. Here we do not explicit specify property P as a function to Booleans over the content of the array, but take an abstract view that the array itself contains Booleans. The index of the array starts from 1. The size of the array is represented by constant M and it must be positive.

constants: $ARRAY, M$	axioms: axm0_1 : $M \in \mathbb{N}_1$ axm0_2 : $ARRAY \in 1..M \rightarrow \text{BOOL}$

The initial model contains only one integer variable called *result*. The invariant just specifies the type of the variable (integer) and initially, the variable can take any random value.

	invariants: inv0_1 : $result \in \mathbb{Z}$	init begin $result := \mathbb{Z}$ end
variables: $result$		

There is only one event **final** (beside the initialisation) to specify the result of the program *in-one-shot*. The aim of the program is encoded in the guard as constraints for parameter k . The first guard states that k is within the domain of the array or has the value $M + 1$. The second guard states that any element of the array with the index strictly less than k has the value **F**. The last guard state that if k is not $M + 1$ (i.e., k is within the domain of the array) then the value of the array at index k is **T**. The action of the event just assigns k to the variable *result*.

```

final
  any  k  where
     $k \in 1 \dots M + 1$ 
     $\forall j \cdot j \in 1 \dots k - 1 \Rightarrow ARRAY(j) = \mathbf{F}$ 
     $k \neq M + 1 \Rightarrow ARRAY(k) = \mathbf{T}$ 
  then
     $result := k$ 
end

```

This event models *what* happens at the end of the program without specify exactly *how* the result is achieved. This is a typical strategy in Event-B for starting the development with an abstract specification and subsequently refine this specification to obtain a program which is correct by construction.

First Refinement

The first refinement introduces the idea of using two processes. Here the context needs to be extended to include the notion of two different non-empty parts of the array. Two constants, namely *PART1* and *PART2*, are introduced with properties that they are non-empty and cover the domain of the array (but not necessarily to be disjoint).

constants: <i>PART1</i>, <i>PART2</i>	axioms: axm1_1 : $PART1 \cup PART2 = 1 \dots M$ axm1_2 : $PART1 \neq \emptyset$ axm1_3 : $PART2 \neq \emptyset$

At this point, the necessary information about the two sub-processes in order to obtain the final result of the program is: whether or not they already terminate, and the published results of the two processes. They are represented by a pair of variables, namely *finish1* and *publish1* (resp. *finish2* and *publish2*) for *process₁* (resp. *process₂*). Initially *finish1* (resp. *finish2*) is given the value **F**, i.e, the process has not yet terminated; and *publish1*

(resp. *publish2*) is assigned the value $M + 1$, i.e., the process has not yet found any result.

variables: $\dots, finish1, publish1, finish2, publish2$

Before stating the invariants related to these new variables, we first look at the refinement of **final** event with the new set of variables. This event is carried out when the two processes have finished and the result taken is just the minimum of the two published values.

```

final
  refines final
  when
    finish1 = T
    finish2 = T
  with
    k = min({publish1, publish2})
  then
    result := min({publish1, publish2})
  end

```

In order to prove the refinement of the **final** with respect to its abstract version, we need to give a witness for the disappearing parameter k of the abstraction. Here the parameter k is exactly the minimum of the two published values. Given the witness, the *simulation* proof obligation becomes trivial since both the abstract and concrete events assign equivalent expressions to the variable *result*.

We still need to prove *guard strengthening*. Using the information of the witness, what we need to prove is just the abstract guards where parameter k is substituted with $\min(\{publish1, publish2\})$.

$$\begin{aligned}
& \min(\{publish1, publish2\}) \in 1 \dots M + 1 \wedge \\
& (\forall j \cdot j \in 1 \dots \min(\{publish1, publish2\}) - 1 \Rightarrow ARRAY(j) = F) \wedge \\
& (\min(\{publish1, publish2\}) \neq M + 1 \Rightarrow \\
& \quad ARRAY(\min(\{publish1, publish2\})) = T)
\end{aligned}$$

This requires us to give some invariants for the newly introduced variables. Note that the predicate $publish1 \neq M + 1$ means that *process*₁ has already published some results.

invariants:

- inv1_1** $publish1 \neq M + 1 \Rightarrow finish1 = \top$
- inv1_2** $publish1 \neq M + 1 \Rightarrow publish1 \in PART1$
- inv1_3** $publish1 \neq M + 1 \Rightarrow ARRAY(publish1) = \top$
- inv1_4** $publish1 \neq M + 1 \Rightarrow$
 $(\forall i \cdot i \in PART1 \wedge i < publish1 \Rightarrow ARRAY(i) = \text{F})$
- inv1_5** $finish1 = \top \wedge publish1 = M + 1 \Rightarrow$
 $(\forall i \cdot i \in PART1 \wedge i < publish2 \Rightarrow ARRAY(i) = \text{F})$
- inv1_6** $publish2 \neq M + 1 \Rightarrow finish2 = \top$
- inv1_7** $publish2 \neq M + 1 \Rightarrow publish2 \in PART1$
- inv1_8** $publish2 \neq M + 1 \Rightarrow ARRAY(publish2) = \top$
- inv1_9** $publish2 \neq M + 1 \Rightarrow$
 $(\forall i \cdot i \in PART2 \wedge i < publish2 \Rightarrow ARRAY(i) = \text{F})$
- inv1_10** $finish2 = \top \wedge publish2 = M + 1 \Rightarrow$
 $(\forall i \cdot i \in PART2 \wedge i < publish1 \Rightarrow ARRAY(i) = \text{F})$

The invariants are symmetric for $process_1$ and $process_2$, hence we only give the description of the first five invariants associated with $process_1$ here.

inv1_1 states that if $process_1$ has published some result then it must have terminated. This also means the process can publish at most once.

inv1_2–inv1_4 states that $process_1$ *cannot lie*: if it publishes some result then this must be the smallest index that it can find within $PART1$.

inv1_5 states that in the case where $process_1$ terminates without publishing any values, it has given up because it cannot find any better result than the other process $process_2$. Considering the two possibilities for $process_1$ to terminate:

- it has searched all the indices in $PART1$ and did not find any result, or
- it looks at the published value of the $process_2$ and know that it cannot find a better (smaller) result.

In both situations, the invariant holds trivially.

We now abstractly construct the events to model the effect of the two processes on the new variables. These events correspond to the two cases in which a process can terminate. Here, we consider the events corresponding to $process_1$ only.

The first case is when $process_1$ finds a result within $PART1$ and terminates. Here $publish1 = M + 1$ is a theorem which is the consequence of the

first guard $finish1 = FALSE$ and invariant **inv1_1**. The other case is when $process_1$ terminates without publishing any value.

```

found_1
  any  k  where
    finish1 = F
    k ∈ PART1
    ARRAY(k) = T
    ∀ i · i ∈ PART1 ∧ i < k ⇒ ARRAY(i) = F
    publish1 = M + 1
  then
    finish1, publish1 := T, k
  end

not_found_1
  when
    finish1 = F
    ∀ i · i ∈ PART1 ∧ i < publish2 ⇒
      ARRAY(i) = F
  then
    finish1 := T
  end

```

Decomposition

In the previous refinement step, we introduced the *interface* of the processes, i.e. the shared variables and events describing how these variables can be changed, which guarantees the correctness of the program. At this point, we want to develop in details each process independently. We apply the technique of decomposition (shared variable) as described earlier in Section 8.1.1. There will be three different processes: *main* (final), $process_1$ (found1, not_found1) and $process_2$ (found2, not_found2).

As a result, we have three different sub-models, one for each process. Amongst these sub-models, the development *main* is straightforward and is not of our interest here. We concentrate on the sub-model for $process_1$ ($process_2$ is symmetric).

The sub-model for $process_1$ contains three shared variables: *finish1*, *publish1* and *publish2* and no private variables. This process does not refer to either *result* (the global result) or *finish2* (if the other process has finish or not).

variables: *finish1, publish1, publish2*

According to the event distribution, this model of $process_1$ has two internal events, namely `found_1` and `not_found_1`, which are the exact copy of the original events. The other events become external which need to be generated as follows. We present the original events on the left and the corresponding external events for $process_1$ on the right.

final when $finish1 = T$ $finish2 = T$ then $result := \min(\{publish1, publish2\})$ end	(ext_)final any $finish2$ where $finish1 = T$ $finish2 = T$ then SKIP end
found_2 any k where $finish2 = F$ $k \in PART2$ $ARRAY(k) = T$ $\forall i \cdot i \in PART2 \wedge i < k \Rightarrow$ $ARRAY(i) = F$ $publish2 = M + 1$ then $finish2, publish2 := T, k$ end	(ext_)found_2 any $k, finish2$ where $finish2 = F$ $k \in PART2$ $ARRAY(k) = T$ $\forall i \cdot i \in PART2 \wedge i < k \Rightarrow$ $ARRAY(i) = F$ $publish2 = M + 1$ then $publish2 := k$ end
not_found_2 when $finish2 = F$ $\forall i \cdot i \in PART1 \wedge i < publish2 \Rightarrow$ $ARRAY(i) = F$ then $finish2 = T$ end	(ext_)not_found_2 any $finish2$ where $finish2 = F$ $\forall i \cdot i \in PART1 \wedge i < publish2 \Rightarrow$ $ARRAY(i) = F$ then SKIP end

In the next coming sections we focus on the further development of $process_1$.

Further (sub-)refinements

In this section, we present the sketch of the further development of $process_1$. The refinement steps are all typical super-position refinement where more details about the actual process are introduced at each step as mentioned early

in the beginning of Section 8.1.3. We do not present in detail the proofs of correctness of the refinement steps here.

Introducing the local index In the first sub-refinement for $process_1$, we introduce the index that the process is currently checking. This is represented by the new variable $index1$. The following invariants state that this process investigates only the part of the array belongs to $PART1$ in ascending order and it cannot skip any index.

variables: $\dots, index1$

The invariant constraints the newly introduced variable is as follows:

invariants:

inv2_1 $index1 \neq M + 1 \Rightarrow index1 \in PART1$

inv2_2 $\forall k \cdot k \in PART1 \wedge k < index1 \Rightarrow ARRAY(k) = F$

The first invariant states that $index1$ is either not in the domain of the array (i.e., $M + 1$) or within the part belongs to $process_1$. The second invariants state that the value of the array at any index smaller than $index1$ and in $PART1$ is F, this is because $process_1$ must have already checked the value at those indices before moving to the new index.

The internal event `not_found_1` is unchanged. It trivially maintains the new invariants since it only modifies variable $finish1$. The same applies to external events, i.e., `(ext_)final`, `(ext_)found_2`, `(ext_)not_found_2` (which are always unchanged during refinement), since they do not refer to variable $index1$.

We now refine the internal event `found_1` to use $index1$; and introduce a new event `inc_1` to model the case where the value at the current index is F, hence $process_1$ moves to the next index.

```

found_1
  refines found_1
  when
    finish1 = F
    index1  $\neq$  M + 1
    ARRAY(index1) = T
  with
    k = index1
  then
    finish1, publish1 := T, index1
  end

```

```

inc_1
  any  $i$  where
     $ARRAY(index1) = F$ 
     $i \neq M + 1 \Rightarrow i \in PART1$ 
     $index1 < i$ 
     $\forall j \cdot j \in PART1 \wedge index1 < j \Rightarrow i \leq j$ 
  then
     $index1 := i$ 
  end

```

For event **found_1**, the information from the witness $k = index1$ and the two invariants declared above guarantees that this is a correct refinement of the abstract event. For event **inc_1**, the parameter i is the smallest index in $PART1$ which is greater than $index1$, or $M+1$ if such an index does not exist. The proof that this event maintains the invariants is intuitive and can be found in our on-line archive [Hoa09]. With the information from the witness $k = index1$, the proof obligations for simulation and invariant preservation are trivial since the expressions assigned to the common variables $finish1$ and $publish1$ are the same. For the guard strengthening proof obligation we must prove the following, again using the witness information.

- $index1 \in PART1$. But since we have the second guard $index \neq M + 1$ and together with invariant **inv2_1**, we have $index1 \in PART1$.
- $ARRAY(index1) = T$ is the last guard.
- $\forall i \cdot i \in PART1 \wedge i < index1 \Rightarrow ARRAY(i) = F$ is exactly the same as the invariant **inv2_2**.

Introduce the read value In this refinement, we introduce the read value of process represented by variable $read1$. The constraint for this variable is expressed by invariant **inv3_1**: its value is either $M + 1$ or the published value of the other process, i.e., $publish2$. A new event **read_1** is introduced to model the situation when $process_1$ reads the published value of $process_2$. This event sets the value of $read1$ to $publish2$ hence clearly maintains the invariant **inv3_1**.

variables: $\dots, read1$	invariants: inv3_1 $read1 \neq M + 1 \Rightarrow read1 = publish2$

```

read1
  begin
     $read1 := publish2$ 
  end

```

The only change to event **inc1** is two extra guards: $index1 < read1$ and $index1 < publish1$. Since this event does not change variables $read1$ and $publish2$, it preserves the invariant **inv3_1** trivially.

The event **found_1** is refined by replacing the guard $index1 \neq M + 1$ with the following two guards: $index1 < read1$ and $index1 < publish1$. Since both $publish1$ is either $M + 1$ or belongs to $PART1$, $publish1$ is no greater than $M + 1$. Together with the guard $index1 < publish1$, $index1$ is strictly smaller than $M + 1$ hence the proof obligation for guard strengthening holds trivially.

We refine the remaining internal event **not_found_1** by replacing the guard

$$\forall i \cdot i \in PART1 \wedge i < publish2 \Rightarrow ARRAY(i) = F$$

with

$$index1 < read1 \Rightarrow publish1 \neq M + 1.$$

We do not go into detail of the proof why this is a correct guard strengthening, but refer the readers to our machine checked model on the web [Hoa09].

For the external events, even though they are not refined, we need to prove that they maintain the invariant **inv3_1**. In this case, we need to consider those events that modify variable $publish2$. In our development, this is event **(ext_)found_2**. The important part for our proof in this event is the theorem in the guard, i.e., $publish2 = M + 1$, and the action $publish2 := k$. According to the action, we have to prove that $read1 \neq M + 1 \Rightarrow read1 = k$, under the assumption of the invariants and the guards. From the theorem in guard $publish2 = M + 1$ and invariant **inv3_1**, we have $read1 = M + 1$ (since if it is not, then we have $publish2 = read1 \neq M + 1$). Hence $read1 \neq M + 1 \Rightarrow read1 = k$ holds trivially.

Introduce the address counter In this last sub-refinement of $process_1$ we introduce the address counter in order to obtain the unfolded program as described in Section 8.1.2. The address counter is represented by the variable $address1$ and has value within the range $1 \dots 3$. Moreover, the value of the address counter is 3 when the process has finished.

variables:	$\dots, address1$	invariants:	$inv4\_1 \quad address1 \in 1 \dots 3$
			$inv4\_2 \quad finish1 = TRUE \Leftrightarrow address1 = 3$

Here, beside the trivial addition of the check for the value address counter to guards of the events and the modification of the address counter to the actions of those events, we also refine the guard to match with the actual algorithm.

- In events `found_1` and `inc_1`, two guards $index1 < publish1$ and $index1 < read1$ are replaced by guard $index1 < \min(\{publish1, read1\})$. The guard strengthening argument bases on trivial arithmetic reasoning.
- For event `not_found_1`, the abstract guard $index1 < read1 \Rightarrow publish1 \neq M + 1$ is replaced by $\neg(index1 < \min(\{publish1, read1\}))$. The argument for guard strengthening is as follows. We have to prove that $\neg(index1 < \min(\{publish1, read1\}))$ and $index1 < read1$ then $publish1 \neq M + 1$. From $\neg(index1 < \min(\{publish1, read1\}))$, we have either $index1 \geq read1$ or $index1 \geq publish1$. However, since $index1 < read1$, it must be the case that $index1 \geq publish1$, hence $publish1 < read1$. From the invariant **inv3_1** states that $read1 \neq M + 1 \Rightarrow read1 = publish2$, we can deduce that $read1 \leq M + 1$. Since $publish1$ strictly smaller than $read1$, $publish1 \neq M + 1$.

The resulting internal events are as follows. These events conform with the notion of atomicity mentioned earlier. They correspond to the pseudo-code as of the unfolded process in Section 8.1.2.

```

read1
  when
    address1 = 1
  then
    address1, read1 := 2, publish2
  end

not_found_1
  when
    address1 = 2
     $\neg(index1 < \min(\{publish1, read1\}))$ 
  then
    address1, finish1 := 3, T
  end

found_1
  when
    address1 = 2
     $index1 < \min(\{publish1, read1\})$ 
     $ARRAY(index1) = T$ 
  then
    address1 := 3
    finish1 := T
    publish1 := index1
  end

```

```

inc_1
  any i where
    address1 = 2
    index1 < min({publish1, read1})
    ARRAY(index1) = F
    i ≠ M + 1 ⇒ i ∈ PART1
    index1 < i
    ∀ j · j ∈ PART1 ∧ index1 < j ⇒ i ≤ j
  then
    address1, index1 := 1, i
  end

```

Proof Statistics

The proof statistics for the development is in the table below. We only take into account the number of obligations for sub-refinement models once, since the refinements for both process $process_1$ and $process_2$ are symmetric. We can use techniques such as pattern or generic instantiation in order to reuse the sub-development without reproving again. In table, 50% of the proof obligations are in the model before decomposing. This indicates that this refinement is the most important and difficult step in our approach.

The most number of proof appear in the fourth refinement when the address counters are introduced even though most of them are discharged automatically. Perhaps not surprisingly, the most difficult proofs appear in the fifth refinement when we reason about the “final” events. It is the core of the algorithm where we have to reason about the interaction of the two parallel processes in order to guarantee the correct final result.

Model	Number POs	Auto.(%)	Manual (%)
Initial context	0	0 (N/A)	0 (N/A)
Initial model	3	3 (100%)	0 (0%)
First extended context	0	0 (N/A)	0 (N/A)
First refinement	46	44 (96%)	2 (4%)
First sub-refinement	14	10 (71%)	4 (29%)
Second sub-refinement	6	5 (83%)	1 (17%)
Third sub-refinement	22	16 (73%)	6 (27%)
Total	91	78 (86%)	13 (14%)

8.1.4 Related Work

The problem of verifying the *FindP* program has been tackled using different methods, notably using Owicki/Gries' *interference-free* [OG76] and Jones' *rely/guarantee* approach [Jon83b, Jon85]. Moreover, the *FindP* program has been used as an illustrated example for the formalisation of these two approaches in Isabelle/HOL [Nie01].

The work of Owicki/Gries [OG76] extends Hoare's deductive system for sequential programs [Hoa69] in order to prove the correctness of parallel programs. Their proofs of correctness for parallel statements centre around the notion of *interference-free* which is defined as follows. Given a proof of Hoare's triple $\{P\} S \{Q\}$ and a statement T with precondition $pre(T)$, T does not interfere with $\{P\} S \{Q\}$ if

InfFree1 $\{Q \wedge pre(T)\} T \{Q\}$, i.e., T maintains the post-condition Q , and

InfFree2 for any sub-statement S' of S , $\{pre(S') \wedge pre(T)\} T \{pre(S')\}$.

Within our approach, the above two conditions are verified during the development of the model at various refinement levels. At the abstract level before decomposing, S and T are some events of the models and the post-condition Q are just some invariants. For example, S are some events belonging to $process_1$ and T are events belonging to $process_2$, Q are the invariants that state the outcome of $process_1$, e.g. **inv1_1–inv1_5**. We have to prove that these invariants are maintained by any events T and this corresponds to condition **InfFree1**.

Furthermore, during the sub-refinement of a process, sub-statements S' of S are introduced. At the same time, new invariants are added and these invariants correspond to the preconditions $pre(S')$ in the proof of $\{P\} S \{Q\}$ using Hoare's deductive system. Hence the condition **InfFree2** is verified by proving that events T (now becoming external events) maintain the new invariants.

This is somewhat not surprising, since in our approach, the role of external events is to keep the information about the possible changes on shared variables by different processes. During the refinement of a sub-process, we need to take into account the effect of these external events so that they do not “interfere” with the development of this sub-process. The main advantage of our approach over the work from Owicki/Gries is that these external events are at the abstract level rather than concrete statements as defined in the *interference-free* conditions. This reduces the complexity of the verification process.

Comparing to the Owicki/Gries approach, our method is closer to the *rely/guarantee* approach of Jones [Jon83b]. The approach extends the notion

of Hoare's triple $\{P\} S \{Q\}$ to encode the rely condition R and guarantee condition G . By definition, a condition $\{P, R\} S \{G, Q\}$ is satisfied by S if: under the assumptions that S starts in state satisfies the precondition P , and any external transition satisfies the rely condition R ; then S ensures that any internal transition of S satisfies the guarantee condition G , and if S terminates then the final state satisfies postcondition Q .

We focus on an example rule for parallel composition.

$$\begin{array}{ll}
R \vee G_1 \Rightarrow R_2 & \textbf{(RG1)} \\
R \vee G_2 \Rightarrow R_1 & \textbf{(RG2)} \\
G_1 \vee G_2 \Rightarrow G & \textbf{(RG3)} \\
\{P, R_1\} S_1 \{G_1, Q_1\} & \textbf{(RG4)} \\
\{P, R_2\} S_2 \{G_2, Q_2\} & \textbf{(RG5)} \\
\hline
\textbf{PAR-I} \quad \{P, R\} S_1 \parallel S_2 \{G, Q_1 \wedge Q_2\} &
\end{array}$$

The rule is interpreted as follows. Statement $S_1 \parallel S_2$ satisfies $\{P, R\} S_1 \parallel S_2 \{G, Q_1 \wedge Q_2\}$ if the following conditions are met. Firstly, both “global” rely condition R and the guarantee condition of one statement ensure the rely condition of the other (**RG1** and **RG2**). Secondly, both guarantee conditions of the two statements ensure the global guarantee condition G (**RG3**). Lastly, S_1 and S_2 independently satisfy their corresponding rely/guarantee condition (**RG4** and **RG5**)

Note that both rely and guarantee conditions are relations over two states. They are indeed similar to events in Event-B which correspond to a relations over pre-/post-states. Moreover, the implication between rely/guarantee conditions is the same as event refinement. Within our approach, a pair of internal/external events encodes rely/guarantee conditions where the rely condition corresponds to the external event and the guarantee condition corresponds to the internal event. The generation of external events guarantees that they are the abstractions of the corresponding internal events. In fact, our generation of sub-models as described in Section 8.1.1 guarantees that the resulting sub-models satisfy the parallel composition rule. This is the advantage of our approach over *rely/guarantee* method. In fact the external events are the strongest possible condition that the other process can rely on. In practise, the rely/guarantee conditions could be more abstract, e.g. requires only that the value of some variables decrease monotonically [Jon07]. Moreover, rely/guarantee is usually used for composition rather than decomposition as in [AL95].

The decomposition technique also appears in many other approaches, with similar intuition: Breaking a specification into smaller pieces and reasoning about them independently. For example, in the work of Abadi/Lamport

[AL95], this is captured by their *Decomposition Theorem* and a generalised version of it. The most important idea in their approach is to find some properties E (also called *environment*) of the other processes assumed by a process. However, in another study, Lamport claimed that decomposition might not be that useful [Lam97]. One of the argument is the difficulty in inventing the *environment* properties and checking the hypotheses of the decomposition theorem. In our approach, we *derive* these properties from the overall purpose of the program using refinement (step 2 of our approach). This is also the reason why we consider the class of parallel programs that achieve some intended result.

Given the close relationship between the Action System formalism [Bac90] and Event-B, not surprisingly, stepwise refinement has been considered for developing parallel systems in Action System in early work of Back/Sere [BS89, BS91]. The shared variable decomposition in Event-B corresponds to their notion of *concurrent action system* (in contrast to *distributed action system* with shared actions). However, the approach presented in [BS89] based on the notion of refining atomicity introduces the notion of parallelism quite late in the development (almost as the last step of the refinement chain). The reason for this delay is that the decision for implementing the system as concurrent action system or distributed action system can be made as late as possible. In our example, we have this decision of using shared variables in advance, hence we can take the advantage of having the decomposition early to reduce the complexity. We consider the use of shared variables as a part of the design process of the program rather than an implementation detail.

8.1.5 Conclusion

We have presented a method for developing parallel programs using refinement and decomposition techniques. Refinement gives us the possibility to abstractly define the aim of the programs which helps us to understand the purpose of these programs. Decomposition allows us to reduce the complexity of the development by separately developing sub-processes while keeping track of minimum information on what other processes can do. Our approach should be applicable to all programs that use several parallel processes in order to obtain a certain goal.

Our approach introduces the possible *interaction* between processes early in the development in order to take the advantage of decomposition. This is different from the approach where one develops processes according to the implementation of the process with possible *cheating* (e.g. one process directly looks into the value of the other process), and subsequently refines the model until there is no more cheating. This approach has been proposed

in [Abr10] and is used in many other examples. Applying this approach without using decomposition, the two processes are developed together, hence the development also has higher complexity comparing to our approach.

We have shown here a concrete development as an application of our proposed approach. However, in reality, we start with a model where all processes are developed together. This results in a high number of proof obligations, even though they are not too difficult to discharged. The reason is that these processes are developed together hence with respect to reasoning about “interference” of one process on the other, we have to reason also at the concrete level of the interfering process, rather than its abstraction.

The key point in our development using decomposition lies in the model that is being decomposed, where we have to abstractly specify the effect of the two future processes on shared variables. We use the overall intended result of the program to help us to *derive* the requirement on the future processes.

Furthermore, as a result of using step-wise refinement, we can develop sub-processes using different implementations as long as they satisfy the abstraction. As an example, we can also “implement” the two processes (inefficiently) by not checking the published values of the other processes or having more fine-grained version of atomicity.

For future work, we would like to apply our method to other standard parallel programs (not necessarily ones with intended result) known from literature, such as “bounded buffer”, “partition of set” or “bubble-lattice sort”, which have been studied using other approaches [Bar85]. Our approach should not only be used for verification a posteriori but also for finding proofs of correctness for such systems.

8.2 Rely/guarantee and Event-B

Event-B has a clear notion of “event decomposition” that *can* be used to develop concurrent algorithms. There are of course many other ways of reasoning about concurrency including Concurrent Separation Logic [O’H07]. One interesting comparison is with the rely/guarantee approach and the comparison is made even more enlightening because of the approach outlined in the previous section that looks at how some form of rely/guarantee idea can be added to event refinement.

First, some brief notes on rely/guarantee thinking and references to further sources. Jones’ thesis [Jon81] was followed by two shorter papers [Jon83a, Jon83b]; the most accessible brief exposition is probably [Jon96]; a more recent paper on semantics is [CJ07] and a challenging application in [JP10].

There is a wide literature on rely/guarantee including some interesting combinations with Separation Logics.

The basic idea of is very simple. Just as few programs can run in arbitrary states (and we record the limit their applicability with pre conditions), almost no concurrent programs can achieve a useful result under arbitrary interference. A rely condition is a relation that records the limit of acceptable interference. Guarantee conditions are just the other side of the coin: they record the maximum interference the code can generate.

In [HA10], a concerted effort is made to add rely/guarantee thinking to Event-B. Not surprisingly, this graft is difficult. An essential tenet of “action systems” and Event-B that events are atomic. In contrast, rely/guarantee is aimed at interference *during* an operation. The development in [HA10] certainly has to generate more cumbersome interface considerations than, for example, in [Jon81, CJ07] but it is the fundamental problem of atomicity that makes the former difficult.

That having been said, [HA10] provides a valuable point of comparison.

Chapter 9

Code Generation

M. Butler, F. Degerlund, A. Edmunds, A. Rezazadeh

Code generation is an important part of the formal engineering tool chain, which enables complete support for development from high-level models down to executable implementations. We address the need for specification of, and code generation for, sequential and concurrent implementations. Our work in DEPLOY has a particular emphasis on real-time embedded software. This chapter covers two main approaches to code generation from Event-B models. In the first approach (Section 9.1), Event-B is augmented with sequential control structures that guide the structure of the generated code. In effect the modeller is defining the scheduling of events through a collection of concurrent sequential programs. In the second approach (Section 9.2), the scheduling of events is done in an interleaved manner over a several processors so that the modeller does not provide explicit scheduling.

9.1 Code Generation with Tasking Event-B

In previous work, we described an approach for code generation from Event-B, tailored to Java implementations of concurrent programs. The experience gained in that undertaking informs our new approach. In particular, our interest has now turned towards the multi-tasking software used with embedded control systems. We wish to generate Ada and C code, since these languages are often used to implement these systems. We have developed a new methodology, and extended the existing Event-B tool to facilitate code generation. The approach has been formulated so that it integrates well with the existing Event-B methodology and tools. Event-B is

extended with AutoTask, Shared, and Environ machines. We use these features to specify how elements should be translated to code, i.e. controller tasks, protected objects and environment tasks. Using refinement, decomposition, and the extension, we structure projects so that they are amenable to automatic code generation. This necessitates the introduction of a great deal of low-level detail into the specification; so models grow in size considerably. To ensure models remain tractable, we decompose the models (perhaps a number of times). Decomposition encourages modular reasoning, and makes proof more tractable. AutoTask machines contain a behavioural description; making use of imperative, programming-like constructs. We also specify a task's priority and its life-cycle, i.e. whether it is periodic, triggered, repeating or one-shot. However, timing aspects are not modelled formally. The interaction between controller tasks and the environment, and controller tasks and protected objects, is described by the synchronized events that arise during model decomposition. These events can have one of several roles. They may participate in a procedure call to a protected object, or they may take part in a sensing, or actuating, role - interacting with the environment. In the generated code, sensing and actuating may be simulated, using a type of subroutine call; or additional information can be provided to allow a task to read/write directly from/to a specified memory location. To validate the approach we undertook a case-study, which we use as a running example.

9.1.1 Introduction

In this section we focus on the domain of multi-tasking, embedded control systems. Our interest is the application of techniques, and provision of tools, for modelling the systems, and generating code from the models. In previous work [Edm10, EB08, EB10] we described an intermediate language for generating Java code [GJSB04] for multi-tasking implementations, using an object-oriented intermediate specification. During our investigations, we encountered difficulties due to the large semantic gap between Event-B and the intermediate specification language. We also found that models were intractable due to large size of the models. We found that our language introduced too much fine-grained atomicity; which gave rise to a large number of proof obligations.

To address the problems encountered in the previous work, and advance the code generation techniques, we developed a methodology which is more closely integrated with the existing Event-B tools, and approach. In order to achieve a small semantic gap we make just a few additions to the Event-B language. To address the problem of intractably large models we use decomposition [But09a, SPHB10]. When we are ready to provide implementation

details, we use the constructs introduced in the extension (we call it Tasking Event-B) to annotate the development. The annotations and Event-B models are then used to generate code. Tasking Event-B is under-pinned by Event-B operational semantics, so we can also generate a model of the implementation, and show that it is a refinement of the development. We have developed a demonstrator tool [Depb] to validate the approach; the tool integrates with the existing Rodin platform [RODc]. We illustrate the approach using a case study with an embedded Heater Controller, and we use of Ada [Bar06] as the target programming language. Our approach, however, is equally applicable to other programming languages, such as C [KR88].

We continue with subsection 9.1.1 in which we discuss our motivation; and subsection ?? provides an overview of the existing Event-B approach. subsection 9.1.2 describes how we model and implement tasks and protected objects. subsection 9.1.3 describes our approach to generating controller tasks modelling and simulating the environment. In subsection 9.1.4 we present a case study. subsection 9.1.5 describes how we can read/write directly to memory. subsections 9.1.6 and 9.1.7 discuss some theoretical aspects of implementing the model. subsection 9.1.8 is a brief résumé of the code generation step. subsection 9.1.9 provides a summary and discussion.

Motivation

A number of the DEPLOY industrial partners have an interest in the development of multi-tasking, embedded control systems, to which the Event-B method has been applied. The modelling activity is beneficial, leading to a precise definition of the system under development, and shows that various properties of the system hold true as the development proceeds. The work on the final step of generating code, which is required to obtain the full benefit of the Event-B approach, has progressed gradually during the project life-cycle. The project's industrial partners use a number of programming languages to implement their systems. We chose one language, Ada [Bar06], as a starting point for code generation. However, our approach is not limited to Ada, but we consider it to be a good choice because it is typically used in safety-critical systems. This is primarily because it is a stable, well structured, well-defined language. There is also a version called SPARKAda [SPA], for high-integrity systems, which is amenable to formal verification. One of our aims is to support compliance with the Ravenscar profile [Bur99]. This will allow us, in the future, to support the multi-tasking version of SPARKAda, called RavenSPARK [BDV04].

The work presented here is informed by our previous experience with code generation, our aim is to find a solution that integrates well with Event-B.

machine HCtrl_M1	event Get_Target_Temperature2
sees HC_CONTEXT	any tm
refines HCtrl_M0	where
variables cttm cttm2 ...	tm $\in \mathbb{Z}$
invariants	tm = cttm
cttm $\in \mathbb{Z}$	then
cttm2 $\in \mathbb{Z}$	cttm2 := tm
...	end

Figure 9.1: Example of Textual Event-B

Having identified Ada as our target language, we added features that would facilitate modelling at the implementation level, and provide the necessary details for code generation. In our current work, we do not formally model all aspects of an implementation, such as timing issues. We do, however, model the behaviour that relates to the control flow, specified in task bodies. For which we provide Event-B semantics. We developed a case study [ERb] of a Heating Controller to validate the code generation approach. The case study is a typical embedded system, where inputs from the environment are received and processed, and may have some effect in the environment caused by its outputs. We say the inputs to the system are sensed, outputs are actuations. We are interested in modelling sensing and actuating in the environment, and we find it useful to separate the system under development into three distinct partitions. In our approach, we distinguish between three kinds of entities in the implementation. We aim to model controller tasks, environment tasks, and protected objects. The controller tasks interact with the protected objects and environment tasks, but there is no direct communication between controller tasks; this is done via the protected objects.

An Event-B Model

A fragment of an Event-B specification is shown in Fig. 9.1. The specification has a number of variable declarations, which are typed in the **invariants** clause. Additional predicates are added to the invariants clause to describe the desired safety properties. The machine invariant is a conjunction of the list of predicates in the invariants clause. The *Get_Target_Temperature2* event declares one parameter *tm* in the **any** clause. The **where** clause con-

tains the event guards, the first clause is a typing predicate for *tm*. The second constrains the value that *tm* can take. The guards describe enabling conditions for an event. Each clause is a predicate over the sets, constants, and variables of the system. All of the guard clauses must be true for an event to be enabled. If an event is enabled, the updates in its action may occur if the event is selected by the environment. Event actions are defined in the **then** clause; they consist of assignments to machine variables; the assignment may be simple, or non-deterministic. The clause in the example consists of a single, simple assignment, where the variable *cttm2* is assigned the value of the parameter *tm*.

Machine Decomposition and Event Synchronization

Decomposition is a technique that is commonly used to handle complexity. There are currently two styles of composition that can be used with Event-B. They are the shared variable [AH07], and shared event [But09a], styles. In our approach, we make use of shared event decomposition. During the decomposition process, we decide where the machine variables will reside in the decomposed model. In the example of Fig. 9.1 we put *cttm2* in one machine, and *cttm* in another. Decomposition gives rise to two machines; each with an event called *Get_Target_Temperature2*. The event guards and actions will differ though, so it is not just a simple copy. Events can be recomposed following decomposition, to restore the original structure. It can be shown that the recomposed machines refine the original abstract machine, and recomposed events refine their abstract counterparts. The decomposed machines are refined by a composed machine [Sil11a]. The composed machine records the decomposition structure, e.g. the relationships between the shared events. The pairs of events, arising from decomposition, are said to synchronize [But06]. The implementation of synchronized events maps naturally to an atomic subroutine, and we use this in our translation to code.

9.1.2 Tasking Event-B: AutoTask and Shared Machines

The Tasking Event-B extension adds a small number of new constructs to Event-B, to facilitate the mapping of Event-B to code. By keeping the number of new constructs to a minimum, and by making use of existing editing tools, we have maintained a small semantic gap between Event-B and the Tasking Event-B specification. The Event-B decomposition approach described in [But09a, SPHB10] is integral to our methodology. During decomposition, we identify the components of the system, and the interaction

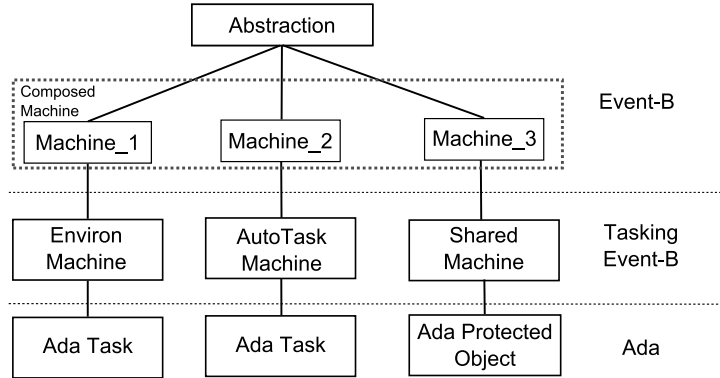


Figure 9.2: Linking Event-B to Code

between them. We then introduce implementation specific constructs, which the translators use to guide code generation. The extended Event-B model also generates a model of the implementation.

Tasking Event-B extends Event-B by introducing the concepts of Auto-Task, Shared, and Environ Machines. Each machine must be annotated with one of the following, to determine its implementation. AutoTask Machines are related to the concept of an Ada [TTBP02] task (but we are not restricted to Ada implementations). Each AutoTask machine models a *controller* task in the implementation, and can be viewed as an abstraction of an Ada task (but we are not restricted to Ada implementations). Shared Machines are related to the concept of a protected resource, such as a monitor [Hoa74], or Ada protected object. Environ Machines model the environment tasks; these will be discussed in the next subsection. Figure 9.2 show the development artefacts, and how they relate to the generated code.

It is possible to generate a simple mapping from Event-B to code, for instance each event can map to a protected procedure. A scheduler can then be used to run the tasks according to a chosen schedule such as [SCZ07]. We take a different approach, where a task body is used to specify the schedule of activities within a task. We also specify the task life-cycle, such as periodicity of a repeating task, using annotations. These annotations are added to Event-B elements, and are used to guide the translation. The translation of an Event-B machine is characterised by one of the following annotations:

- *AutoTask Machine, Environ Machine Shared Machine*

AutoTasks are implemented by *controller* tasks, declared and elaborated in the *Main* procedure of the implementation. In Ada, a task declaration is the specification of a task's interface, and an elaboration is the task's implementation. Both parts may reside in the *Main* procedure. All of the

tasks, generated from AutoTasks, are created when the program first loads, and automatically activated (made ready to run) by the run-time system; before the *Main* procedure body runs (hence the name AutoTasks). When we compile and build the implementation, we identify the *Main* procedure as the program entry point. We only consider AutoTask machines, which are implemented as anonymous tasks of the *Main* procedure. However, in the future we may wish to make use of Ada task types. These are used to create ‘named’ tasks, at compile time; thereby allowing re-use of machine specifications. However, whichever choice we make, fixing the number of tasks, before run-time, is a requirement for compliance with Ravenscar. Therefore, dynamic task allocation is prohibited.

Task Scheduling

The following extensions relate only to AutoTask and Environ machines, and allow a developer to provide implementation details, used by the code generators. Note, however, that timing aspects of periodic tasks, and task scheduling, are not modelled formally in the work that we describe here.

- *TaskType* and *Priority*

The *TaskType* construct is used to define the scheduling, cycle and lifetime of a task. i.e. one-shot, periodic or triggered. The period of a task is specified in milliseconds. *Priority* is an integer value; the task with the highest value priority takes precedence when being scheduled.

Flow Control

The controller tasks of a development are specified using a Tasking Event-B keyword; the machine is annotated with the *AutoTask* keyword. In the generated Ada code, this will be implemented as a controller task, as previously described. Each AutoTask machine has a *task body*. The task body contains flow control (algorithmic) constructs, which are described in the syntax shown in Fig. 9.3. The *sequence* construct is used for imposing an order on events. *branch* is choice between a number of events with mutually exclusive guards. *iteration* indicates that an event repeats until its guard is false. *EventSynch* is used to synchronize an event in an AutoTask machine, with an event in a Shared, or Environ machine. The *EventSync* construct defines an atomic update, between AutoTask and Shared machines; synchronization is implemented as an atomic procedure call. Between AutoTask and the Environ machines, synchronization may be implemented as an atomic *entry*

```

TaskBody ::=
    TaskBody ; TaskBody           // sequence
  |  if EventName                 // branch
    (elseif EventName )*
    else EventName
  |  do EventName                 // iteration
    [finally EventName ]
  |  EventName                   // synchronization
  |  output "String" VarName     // write to console

EventName ::= String
VarName ::= String

```

Figure 9.3: The Task Body Syntax

call, or by direct access to memory. In Ada, a task *entry* is a kind of sub-routine facilitating communication between tasks; it is part of the so-called rendezvous mechanism. It is important to note that, in our approach, controller tasks only communicate with each other through protected objects; however, controller tasks may communicate with environment tasks.

Events

Events can play one of several roles in the mapping to the implementation as follows,

- Synchronization* - the event is one of a pair of events modelling a parametrized procedure.
- Procedure Definition*- the event (on its own) models a non-parametrized procedure.
- Branch* - the event models one part of a branch.
- Loop* - the event models conditional iteration update.

Events can play one of several roles in the mapping to the implementation. They can take part in event synchronization, parameterless procedure call, part of a branch, part of a loop. The Output construct is provided, to allow developers to output text to a console during simulation. We describe the events of an AutoTask as being *local* with respect to the AutoTask machine, and the events of a Shared machine as being *remote* with respect to the AutoTask machine. Events that are local to an AutoTask machine only update the AutoTask machine state; conversely, events that are remote only update

the state of the Shared machine. Synchronised events share parameters to model communication between controller tasks and protected objects, and controller tasks and environment tasks. An Event-B development typically begins with a high-level abstraction of the system; and we refine, and decompose the models, until we are ready to use Tasking Event-B to describe the implementation. Figure. 9.2 shows how a development may proceed from abstract model to implementation. The abstract development may be refined, and then decomposed into separate components. In the Tasking Event-B stage, some of the machines are defined as AutoTask Machines; these will be implemented as Ada tasks and used for deployment. Some are defined as Shared Machines; these will be implemented as Ada protected objects and also used for deployment. The remainder are defined as Environ Machines; these will be implemented as the Ada tasks which can be used to simulate the environment.

Event Synchronization

An example of an AutoTask machine with is shown in Fig. 9.4. The synchronization of the *TCGet_Target_Temperature2* event, with the *SOGet_Target_Temperature2* of Fig. 9.5, models parameter passing between controller tasks and protected objects. The Composed Machine is used to record the synchronizations, and the Code generator uses this to resolve any synchronizations in the task body. In Tasking Event-B we match pairs of Event-B parameters in order of declaration, and map them to parameters in procedure declarations or calls. The controller tasks call protected objects' procedures, so AutoTask event parameters map to actual parameters of the call, and Shared machine event parameters map to formal parameters. Fig. 9.5 shows the *SOGet_Target_Temperature2* event of the Shared machine, it synchronizes with the *TCGet_Target_Temperature2* event of the task shown in Fig. 9.4.

We wish to make a local copy of the Shared machine target temperature *cttm*, in the AutoTask machine, using the assignment *cttm2* := *cttm*. However, in preparation for decomposition, we have ensured that the two variables do not appear in a single action; ultimately, *cttm2* and *cttm* will reside in different machines, so the information will need to be passed as a parameter. For this reason we have specified the event as shown in Fig. 9.1. There is an action *cttm2* := *tm* with guard *tm* = *cttm*. The decomposition of this event can be seen in Figs. 9.4 and 9.5 with the variables in separate machines.

The event parameters play a role in implementation; they map to the actual and formal parameters in the code. The translator derives the imple-

machine Temp_Ctrl_TaskImpl	
is autoTask	event TCGet_Target_Temperature2
refines Temp_Ctrl_Task	refines Get_Target_Temperature2
variables cttm2, ...	any tm
tasktype periodic(250)	where
priority 5	tm $\in \mathbb{Z}$
taskbody is	then
...	cttm2 := tm
TCGet_Target_Temperature2	end
...	

Figure 9.4: Example AutoTask Machine

mented parameters from the machine structure, as a general rule: parameters appearing in actions are *in* parameters, others are out parameters. During translation, an shared machine's event parameters will be mapped to formal parameters in a protected object procedure parameter declaration. A tasking machine's event parameters will map to the actual parameters of a call (after being resolved to a variable) in the case of a synchronization. For translation to code, we match parameters in order of declaration (rather than the names). In the task the assignment $cttm2 := tm$ maps $cttm2$ to an actual outgoing parameter, see Fig. 9.6. In the shared object the tm parameter of the AutoTask event becomes a formal parameter; and the formal parameter is assigned the value of $cttm$ in the call. The guard $tm = cttm$ maps to $tm := cttm$ in the implementation, again, see Fig 9.6. A single event that exists only in the tasking machine will have its action expanded in-line in the implementation, to save the overhead of a call.

9.1.3 Tasking Event-B: AutoTask and Environ Machines

The previous subsection introduced the concept of *AutoTask Machines* and *Shared Machines*. We now introduce modelling of the environment, using the *Environ Machine* annotation. Controller tasks in the system do not communicate with each other; we prohibit inter-task communication because we wish to support implementations that conform to the Ravenscar

```

machine SharedObject1Impl
is sharedMachine
refines SharedObject1
variables cttm ...
invariants ...
events
  ...
  event SOGet_Target_Temperature2
  refines Get_Target_Temperature2
  any tm
  where
    tm  $\in \mathbb{Z}$ 
    tm = cttm
  then
    skip
  end
  ...

```

Figure 9.5: Example Shared Machine

```

s: Shared_Object1Impl;
...
task body Temp_Ctrl_TaskImpl is
    ...
    s.SOGet_Target_Temperature2(cttm2);
    ...
end Temp_Ctrl_TaskImpl;

protected body SharedObject1Impl is
    procedure SOGet_Target_Temperature2
        (tm: out Integer) is
        begin
            tm := cttm;
end;

```

Figure 9.6: Ada Implementation of a Synchronization

profile [Bur99, BDV04]. This profile prohibits the use of the task entries, which are required for inter-task communication. However, we consider that we can relax this restriction, for the *Environ Machines*. This is feasible since the model of the environment will be mapped to simulation code. The remaining deployable code can still be made to conform to the Ravenscar profile, if required. The *Environ Machine* is implemented as an Ada task, and communicates with the deployable tasks using Ada's rendezvous *entry* mechanism. We describe the implementations generated from the Environ machine as environment tasks, and the remainder as controller tasks.

When modelling embedded control systems it is useful to distinguish between the values we are measuring and setting in the environment, and their internal representation in the controllers. We describe the values in the environment as monitored or controlled variables (for sensing and actuating respectively). In the controller tasks, the corresponding stored values are described as sensed and actuated variables.

In a development we use an Environ machine to model the environment. The separation between controller components and environment components is done during decomposition 9.1.1. It is achieved by placing the monitored/controlled variables in one or more machines; and placing the sensed/actuated variables in others. After decomposition, communication between machines is modelled by the synchronization of the events. The implementation of the Environ machine will be somewhat different to that of the Shared machine, seen in the last subsection. The implementation of the latter uses protected procedures as opposed to task entries. A pair of events arises from the process of decomposing a single event (one event will reside in the AutoTask machine, the other in the Environ machine). In order to assist the code generator, we annotate the events in the task, and corresponding event in the Environ machine, to indicate whether they take part in sensing or actuating roles:

- sensing* - the event is one of a pair of events modelling sensing.
- actuating* - the event is one of a pair modelling actuation.

An abstract sensing event can be seen in Fig. 9.7. It is the first refinement from the case study presented in subsection 9.1.4 ; the model has yet to be decomposed, therefore it has no tasking annotations since these are added as the last step of the development. Sensed variable *stm1* keeps track of the monitored temperature *ts1* through the assignment *stm1* := *t1*, and guard *t1* = *ts1* and so on. Note that the guards *t1* = *ts1* and *t2* = *ts2* link the parameters to the monitored variables. Through decomposition, we obtain the synchronizing events that reside in different machines.

```

event Sense_Temperatures
  any t1 t2
  where
    t1  $\in \mathbb{Z}$ 
    t2  $\in \mathbb{Z}$ 
    t1 = ts1
    t2 = ts2
  then
    stm1 := t1
    stm2 := t2
  end

```

Figure 9.7: An Abstract Sensing Event

The *Sense_Temperatures* event is decomposed into two events, one in each of the machines. In preparation for decomposition we named the AutoTask machine *Temp_Ctrl_Task*, and the Environ machine was named *Envir1*. In a subsequent refinement the events were renamed to

TCSense_Temperatures (in the AutoTask machine) and
ENSense_Temperatures (in the Environ machine).

These events can be seen in Fig. 9.8, which show the decomposed events with their Tasking Event-B annotations. The sensed variables *stm1* and *stm2* reside in the AutoTask machine, and can be seen on the left-hand side of the assignment in the *TCSense_Temperatures* event (on the left side of the figure); and monitored variables *ts1* and *ts2* reside in the Environ machine, and appear in the guard of the *ENSense_Temperatures* event. In Event-B decomposition the order of parameter declaration is not important, synchronization is the same regardless of parameter order. However, in Tasking Event-B we introduce a further constraint on parameter declaration to assist with translation. In translating to code we match parameters in order of declaration. Using this constraint simplifies the translation process; see Sects. 9.1.7 and 9.1.7 for more details of the mapping from synchronized events to the implementation.

<pre> event TCSense_Temperatures $\triangleq$ refines Sense_Temperatures sensing any t1 t2 when t1 $\in \mathbb{Z}$ t2 $\in \mathbb{Z}$ then stm1 := t1 stm2 := t2 end </pre>	<pre> event ENSense_Temperatures $\triangleq$ refines Sense_Temperatures sensing any t1 t2 when t1 $\in \mathbb{Z}$ t2 $\in \mathbb{Z}$ t1 = ts1 t2 = ts2 then skip end </pre>

Figure 9.8: Synchronization of a Sensing Event

9.1.4 Case Study

This subsection describes an Event-B development of a simple heating controller. This case study covers the entire development process; starting from a system specification and ending with implementable Ada code. The methodology that we propose here is key to optimising the development process, since the abstract development must be structured in a way that is appropriate for code generation. The development process starts with an abstract specification followed by two successive refinements. We then decompose the model into two separate sub-models, one representing the environment, and the other representing the remainder of the system. The refinement process continues after the first decomposition in order to arrive at a concrete level suitable for implementation. This allows us to derive the appropriate machine structure, needed by the code generation plug-in. The second decomposition gives rise to a number of AutoTask machines, and a Shared machine to manage the protected shared data. The Shared machine synchronizes with the AutoTask machines to model the communication between them. We illustrate how, using the code generation plug-in, a concurrent Ada implementation is generated. The overall aim of this case study is to put in practice the recommended methodological aspects of Event-B, par-

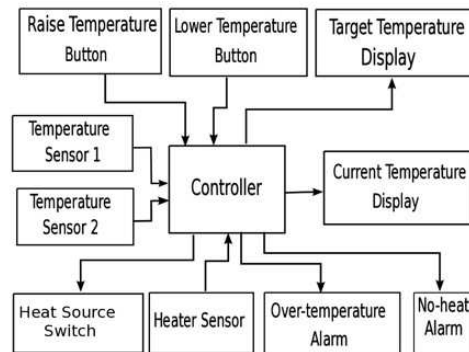


Figure 9.9: Heating Controller Architecture

ticularly those aspects of modelling concerned with decomposition and code generation.

Overview of the System

Figure 9.9 shows an overview of the Heating Controller and related components. The controller in the centre of the diagram communicates with the components in the environment using input and output parameters. At the top there are two buttons that allow the user to increase/decrease Target Temperature. The target temperature periodically will be sent by the controller to the related display to be shown to the outside world. The controller uses two Temperature Sensors to poll the environment temperature. The average of the values read from these two sensors is calculated and displayed by the controller on the Current Temperature Display. If the current temperature is lower than the target temperature, the controller will turn on the heater source using Heat Source Switch, otherwise this switch will be turned off by the controller. The status of the heater itself also will be monitored through the Heater Sensor. If due to some faults the heater is not working properly, the controller will activate either of Over-temperature Alarm or No-heat Alarm.

System Development

Before discussing Event-B models the Heater Controller in more detail, we provide an overview of the development approach in Fig. 9.10. Starting from the top level of the diagram, it shows the most abstract model of the system, or as it is known - the system specification. At the top level we specify the system's main functionality. This includes events for modelling increase/decrease target temperature, polling the environment temperature

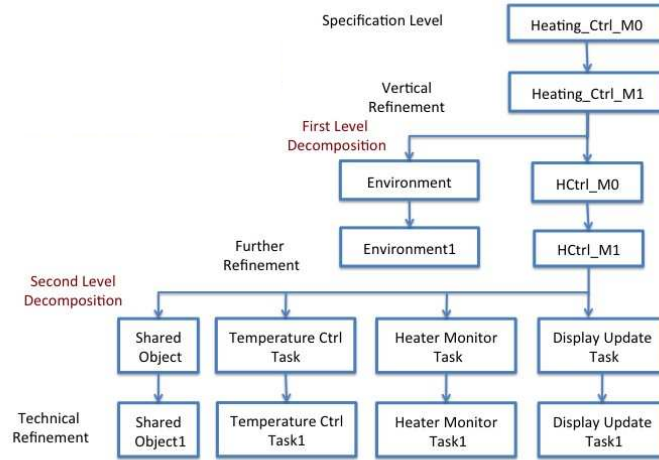


Figure 9.10: The Development Approach

using the two available sensors, calculating the current temperature, turning on/off the heat source, and activating/deactivating the two different alarms. Events at this level are deliberately simple, to keep the specification clear and concise. Event ordering and iteration are introduced in the later stages. In the first refinement we introduce sensing and actuation, which are considered to be design steps. Sensing events model the polling of the state of the increase/decrease buttons, the temperature sensors, and the heater sensor. Actuating events model the updates of target, and current temperature displays. The also model actuation occurring as a result of controller decisions, such as turning the heat on/off, and activating the various alarms.

As illustrated in Fig. 9.10 we decompose our model in two stages. In the first stage we separate the controller, the part of the system that should be implemented, from its surrounding environment. The second stage of decomposition is concerned with the tasking structure of the implementable system. The structure of a particular development is determined by the developer, based on their understanding of the required implementation.

First Decomposition

In Event-B, we usually start modelling by specifying the whole system as a closed system. This includes the system that we are intending to develop and its surrounding environment. Therefore when our model of the system becomes large and complex it is reasonable to separate the controller from its environment. Hence in the first stage of decomposition, we partition our

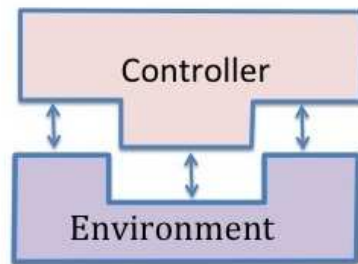


Figure 9.11: Decomposition - Stage 1

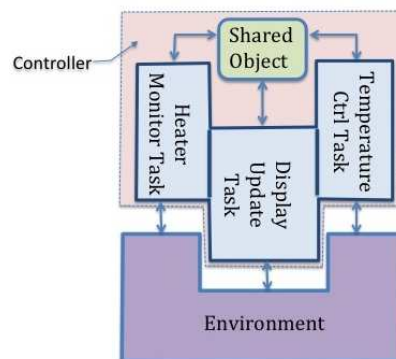


Figure 9.12: Decomposition - Stage 2

model into those two parts, as shown in Fig. 9.11. The environment machine models all of the external parts of the system that the controller interacts with. These include buttons, sensors, actuators and their related behaviours. In addition to modelling updates in the components, the machines modelling the environment and controller, also model the synchronisation between the two.

Second Decomposition

In this, the second stage of decomposition, we decompose the controller into three different tasks which interact with each other through a protected object. This structure is presented in Fig. 9.12. Note that individual tasks are able to communicate with the environment directly, but do not communicate with each other. In practice, the number of tasks, and the way that we distribute different events over these tasks is determined by various factors. Other properties of tasks, such as their priorities; life-cycle (periodic, triggered etc); length of period in milliseconds, if applicable, are defined later using Tasking Event-B.

Tasking Event-B

When we are ready to start adding implementation level details to the model we use Tasking Event-B annotations; entering the *Tasking Event-B* modelling stage shown in Fig. 9.2. We do not present the whole case study specification here, but we use extracts to illustrate the main issues. In the following discussion we use the temperature control task event *Temp_Ctrl_TaskImpl* from Fig. 9.4, as an example. Our main aim is to highlight how we use Tasking Event-B to facilitate the task's interaction with the environment, but we also discuss the task's behaviour in a more general sense. The first step is to use annotations to identify the machines, as one of AutoTask, Environ or Shared machines. When we have identified machines as either Environ, or AutoTask, then we add a task body specification. The task body is used to constrain the Event-B model, in such a way that it can be implemented using commonly available programming constructs, such as sequence, branch and subroutine calls. The generated code can be viewed as an implementation of a schedule of events. Fig. 9.13 shows the *Temp_Ctrl_TaskImpl* task body. The clauses of the task body are labelled, to assist with the explanation. The following list describes the behaviour of the events that they refer to: The

- (1) - a sensing synchronization that models polling of the *ts1* and *ts2* temperature sensors, as shown in Fig. 9.8.
- (2) - an update that calculates, and stores locally, the average of the two temperature values.
- (3) - an actuating synchronization that updates *ctd*, the displayed temperature.
- (4) - a procedure synchronization that gets the target temperature from the protected object.
- (5) - branching choice: set the heater on or off flag in the task.
- (6) - a procedure synchronization that sets the heat source active flag in the protected object.
- (7) - an actuating synchronization that updates *ahsa*, the activate heat source flag.
- (8) - a branching choice: set activate overheat alarm flag in the task.
- (9) - an actuating synchronization that updates *aota*, the activate overheat alarm flag.

task body gives rise to the Ada task body of Fig. 9.14.

The development proceeds by adding annotations to events; we saw annotated sensing events in Fig. 9.8. The *sensing* keyword is used with the *TCSense_Temperatures* and *ENSense_Temperatures* events. This indi-

```

TCSense_Temperatures ;          - - (1) synchs with
                                - - ENSense_Temperatures;

TCCalculate_Average_Temperature ; - - (2)

TCDisplay_Current_Temperature   - - (3) synchs with
                                - - ENDisplay_Current_Temperature;

TCGet_Target_Temperature2 ;     - - (4) synchs with
                                - - SOGet_Target_Temperature2;

if TCTurnON_Heat_Source         - - (5)
else TCTurnOFF_Heat_Source ;

TCSet_Heat_Source_State ;       - - (6) synchs with
                                - - SOSet_Heat_Source_State;

TCActuate_Heat_Source ;         - - (7) synchs with
                                - - ENActuate_Heat_Source;

if TCSwitchOn_OverHeat_Alarm    - - (8)
else TCSwitchOff_OverHeat_Alarm ;

TCActuate_OverHeat_Alarm ;      - - (9) synchs with
                                - - ENActuate_OverHeat_Alarm

```

Figure 9.13: The Temp_Ctrl_TaskImpl Task Body

```

task body Temp_Ctrl_TaskImpl is
    ...
    procedure TCCalculate_Average_Temperature is
    begin
        avt := ((stm1 + stm2) / 2);
    end;
    begin
        ...
        Envir1Impl.ENSense_Temperatures(stm1, stm2);           -- (1)
        TCCalculate_Average_Temperature;                       -- (2)
        Envir1Impl.ENDisplay_Current_Temperature(avt);         -- (3)
        shared_object1implInst.SOGet_Target_Temperature2(cttm2); -- (4)
        if(avt < cttm2) then                                     -- (5)
            hsc := TRUE;
        else
            hsc := FALSE;
        end if;
        shared_object1implInst.SOSet_Heat_Source_State(hsc);   -- (6)
        Envir1Impl.ENActuate_Heat_Source(hsc);                 -- (7)
        if(avt > Max) then                                         -- (8)
            ota := TRUE;
        else
            ota := FALSE;
        end if;
        Envir1Impl.ENActuate_OverHeat_Alarm(ota);               -- (9)
        ...
    end Temp_Ctrl_TaskImpl;

```

Figure 9.14: Implementation of Temp_Ctrl_TaskImpl Task Body

```

accept ENSense_Temperatures(t1: out Integer; t2: out Integer) do
    t1 := ts1;
    t2 := ts2;
end ENSense_Temperatures;

```

Figure 9.15: Implementation of ENSense_Temperatures

cates that the events model polling of the environment; the *actuating* keyword is similar, except that it indicates that events update values is the environment. Clause (1) in the task body, shown in Fig. 9.13, gives rise to the following Ada program statement:

```

Envir1Impl.ENSense_Temperatures(stm1, stm2);

```

Envir1Impl is the name of the environment task, and *ENSense_Temperatures* is the name of the task entry. The sensed variables *stm1* and *stm2*, appear in the actions of *TCSense_Temperatures*. The action models assignment of return values to the variables. In the translation to Ada the AutoTask event parameters is replaced by machine variables, and the variables are passed as actual parameters in an entry call. The entry is implemented as an Ada *accept* statement, see Fig. 9.15, in the *Envir1Impl*. In the body, statements assign the temperature values *ts1* and *ts2* to the formal parameters *t1* and *t2*. The variables *ts1* and *ts2* were added to the Event-B model, in a refinement step, to represent the monitored values in the environment. The monitored variables appear in the guard of the *ENSense_Temperatures* event. See Fig: 9.8. With Ada's pass-by-result parameter semantics, the assignment statement *t1 := ts1* ensures *stm1* is assigned the value of *ts1* upon return. More details of the translation of synchronized events can be found in subsection 9.1.7.

9.1.5 Writing Directly to Memory Locations

The discussion, so far, has focussed on a simulation of the environment whereby the task communicates with the environment using an entry call. In Ada this is implemented as an entry call to the environment task. It may be the case, however, that the developer can specify some memory locations to read from, and write to, directly. Our approach allows developers to annotate event parameters with the address information. We have introduced the

addr annotation, with which we define an addressed variable. It is used in conjunction with the event parameters and environment machine variables. With *addr* we can specify a memory location and its number base. In the Tasking Event-B specification, on the left of Fig. 9.16, the parameter *t1* is given the address *ef14* in base 16. We can see, on the right, the generated Ada code. The parameter *t1* has been mapped to the integer variable declaration **t1: Integer**. The address of the variable has been set using the following statement, the **pragma Atomic(t1)** statement is used to indicate that any access to *t1* must occur atomically:

```
for t1'Address use System'To_Address(16#ef14#);
pragma Atomic(t1);
```

In Ada this is known as an *address clause*. In Fig. 9.16 we can see the Ada *TCSense_Temperatures* procedure. The variable *t1* appears on the right-hand side of the assignment. When the statement is executed, the value is read from the memory location accessed by *t1*, and assigned to *stm1*. If we were generating C code, we would declare a pointer to integer type,

```
int* t1 = (int*)
0xef14
```

and use an assignment *stm1 = *t1*. We can see that this approach differs from that of the entry-call approach shown earlier; the controller task updates its value without a call to an external, environment entry. Now, since there is no call to an entry (which is atomic) the environment must be responsible for ensuring that sensing events with multiple read actions, and actuating events with multiple write actions, are performed atomically. When using addressed variables, in Tasking Event-B, we can discard the environment tasks, and simply use the controller tasks and protected objects in the deployment. In this case we need not consider the effects of atomicity further. However, we may wish to use the environment model for other purposes, in which case we need to ensure that atomic access is achieved in the code.

In the initial stage of development we are able to simulate interaction with the environment using entry calls to the environment task. It may be the case that later in the development, we choose to read from, and write to, memory directly. To do this we simply add the address information to the relevant variables. We can then choose to continue using the environment simulation, which now writes to memory also; or we may have in mind to use another simulator, or perhaps use the code for deployment. If we choose to use environment simulation using addressed variables, we add address information to the monitored variables in the environment. The translator then generates address clauses, and omits the environment entries. The environment task procedures then update monitored variables, in memory, directly.

<pre> event TCSense_Temperatures $\triangleq$ refines TCSense_Temperatures sensing begin any actualIn addr(16,ef14) t1 actualIn addr(16,ef18) t2 when t1 $\in \mathbb{Z}$ t2 $\in \mathbb{Z}$ then stm1 := t1 stm2 := t2 end </pre>	<pre> task body Temp_Ctrl_TaskImpl is stm1 : Integer := 0; stm2 : Integer := 0; ... procedure TCSense_Temperatures is t1 : Integer; for t1'Address use System'To_Address(16#ef14#); pragma Atomic(t1); t2 : Integer; for t2'Address use System'To_Address(16#ef18#); pragma Atomic(t2); begin stm1 := t1; stm2 := t2; end; ... begin loop delay until nextTime; TCSense_Temperatures; ... end loop; end Temp_Ctrl_TaskImpl; </pre>

Figure 9.16: Addressed Variables: Specification and Implementation

9.1.6 Implementing the TaskBody Construct

We now look at the mapping of the flow control constructs to a programming language abstraction. The abstraction appears in the text as a kind of pseudo-code, and its potential mapping to a programming language, such as Ada, should be obvious. The pseudo-code makes use of Event-B guards and actions, and it represents an intermediate translation step during code generation. The intermediate step is not visible to users, though, so we will say no more about it here.

The task body of an AutoTask machine contains *Control* constructs, such as sequence, branch and iteration; the translation of which is mostly straight forward. Control constructs may contain events, and it will be useful, here, to remind ourselves of some terminology, since we make use of it in the remainder of the subsection. Local events are local to an AutoTask machine, they can only update the task's state; conversely remote events belong to any Environ, or Shared machine, and update that machine's state. Synchronized events' parameters model communication between AutoTask, and Shared machines. In [EB11] we described the synchronization of the two events using the definition 9.1 based on the guarded command language [Dij75]. The left-hand side defines the synchronization $\parallel_e$ operator, it relates the decomposition (on the left-hand side) to the composed event (on the right-hand side).

$$g_l \rightarrow a_l \parallel_e g_r \rightarrow a_r \triangleq g_l \wedge g_r \rightarrow a_l \parallel a_r \quad (9.1)$$

We now wish to work with the machine variables, and event parameters, so we extend the notation to allow the guards and actions to range over ordered sets of parameters, P and Q , of the events of local and remote machines respectively; and sets of variables V and W of the local and remote machines. We can then re-write (9.1) as (9.2).

$$\begin{aligned} g_l(P, V) \rightarrow a_l(P, V) \parallel_e g_r(Q, W) \rightarrow a_r(Q, W) &\triangleq \\ g_l(P, V) \wedge g_r(Q, W) \rightarrow a_l(P, V) \parallel a_r(Q, W) &\end{aligned} \quad (9.2)$$

We make use of the parameter definitions, in the translation of the EventSynch construct, shown in Fig. 9.17. The actual parameters of the procedure calls are $p \in P$, the formal parameters used in the subroutines are $q \in Q$. In Fig. 9.17 we describe how the control constructs, events, and their guards and actions, relate to the programming constructs. The right-hand side of the table shows the implementation of the control constructs, but the guards and actions are left as Event-B predicates and expression, to be translated at a later stage. More details of the translation, to code, of the synchronization constructs are given in Sect. 9.1.7.

An unconstrained specification and implementation may lead to undesirable behaviour. This may occur in a naive implementation of an event; say the event is translated into a subroutine with a conditional critical region [Han72] which can block the caller if the condition is false. In this case a task, if the task is both caller, and target of the call, may block itself. This is not desirable since the task state is not visible externally and the task would remain blocked. We therefore prohibit the use of a local guard g_l . This can be seen in the redefinition of the synchronization operator (9.3). Here, we have omitted the details of parameters and variables for clarity. The omission of the local guard is relevant in the translation of the *EventSynch* construct in Fig. 9.17.

$$a_l \parallel_e g_r \rightarrow a_r \triangleq g_r \rightarrow a_l \parallel a_r \quad (9.3)$$

The *Branch* and *Loop* constructs are subject to the restriction that guards are defined for the local event only, that is g_l . There is no remote guard, since the generated code would require a protected procedure call for each branching clause. We decided to keep the implementation as simple as possible, and use just local guards which are not subject to interference. This restriction is summarised in (9.4) which shows a simple branch as an example. It makes use of the alternative choice operator, $\square$, where each guard is disjoint, and in order to ensure that the branches cover all possible outcomes, the conjunction should form a tautology. Our tool should produce proof obligations to ensure this, but is not yet implemented. Similar restrictions are applied to the loop construct, thus our loop definition has the same form as the simple branch.

$$g_l \rightarrow a_l \parallel_e a_r \square \neg g_l \rightarrow SKIP \quad (9.4)$$

Parallel to Sequential semantics

An Event-B action with a number of assignments is a parallel composition of those assignments. Translation to Ada involves serialising the assignments. Parallel composition does not readily translate to its serial form without the introduction of some auxiliary variables to hold the values of machine variables on entry to the procedure. Take the parallel assignment $x := y \parallel z := x$. If $x = 1$ initially, and $y = 0$, then we should have $z = 1$ after the update. If we were simply to serialize this $x := y ; z := x$, then we would have $z = 0$. To remedy this we introduce auxiliary variables for any variable appearing on the left-hand side of an assignment, and also on the right-hand side of some other assignment. We then replace variables appearing on the right-hand side of assignments with the auxiliary variable. Since x appears on the

Control	$\langle Control \rangle^T$
$Control1 ; Control2$	$\langle Control1 \rangle^T ; \langle Control2 \rangle^T$
EventSynch $e_l \parallel_e e_r$ where $e_l \parallel_e e_r =$ $a_l \parallel_e g_r \rightarrow a_r$	$a_l();$ call $target.e_r(p_1 \dots p_n)$ Add to task: subroutine $e_l() \{ a_l; \}$ Add to Protected Object: subroutine $e_r(q_1 \dots q_n) \text{when}(g_r) \{ a_r; \}$
$L: \text{do } e_l \parallel_e e_r$ where: $e_l \parallel_e e_r = g_l \rightarrow a_l \parallel_e a_r$	Add to task body: while(g_l) { $a_l; \text{call } e_r();$ } Add to Protected Object: subroutine $e_r() \{ a_r; \}$
$L: \text{if } e_{1l} \parallel_e e_{1r}$ (elseif $e_{il} \parallel_e e_{ir}$)* else $e_{nl} \parallel_e e_{nr}$ $i \in 1 \dots n$ where: $e_{il} \parallel_e e_{ir} = g_{il} \rightarrow a_{il} \parallel_e a_{ir}$	Add to task body: if(g_{1l}) { $body$ } (elseif(g_{il}) { $body$ })* else { $body$ } $body \triangleq$ $a_{il}; \text{call } e_{ir}();$ and in Protected: subroutine $e_{ir}() \{ a_{ir} \}$

Figure 9.17: Flow Control Implementation

Construct	Parameter Mapping
AutoTask machine	actual
Shared Machine	formal
Action	in
Guard	out

Figure 9.18: Parameter Mappings

left-hand side and right-hand side of the clauses, we introduce an auxiliary variable ini_x , and translate the action to $ini\_x := x ; x := y ; z := ini\_x$

9.1.7 Implementing Synchronized Events

Following the decomposition step, we identify how machines will be implemented. They can be annotated to identify them as AutoTask, Shared, or Environ Machines. In the discussion that follows we show how the events of these machines relate to synchronized communication using a subroutine-style call. We then look at how synchronized events relate to implementations using direct memory access.

Synchronization and Shared Machines

In this subsection we describe how *synchronized* events, involving shared machines, map to the implementation. In Tasking Event-B synchronizations are atomic, and this should be enforced in the implementation. To enforce this we map the pair of events to a single procedure in a protected object. The protected object is Ada's mutual exclusion mechanism.

Event Parameters are paired (using order of declaration) between the synchronized machines. We show how (using synchronization) the Event-B parameters, guards and actions provide implementations of update actions; provide formal input and output parameters for the procedure, provide actual input and output parameters for use by the caller of the procedure. Fig. 9.18 shows how the relationships. For instance, parameters of tasks give rise to actual parameters of the procedure call. Parameters that appear in event guards (other than their typing predicates) map to out parameters etc.

To assist with our explanation we remind ourselves of the description of parameters and variables. The local event has an ordered set of parameters P , and a set of variables V . The remote event has an ordered set of Parameters

Q , and set of variables W , and we can write the synchronization as follows,

$$g_l(P, V) \rightarrow a_l(P, V) \parallel_e g_r(Q, W) \rightarrow a_r(Q, W)$$

We use p to denote an event parameter of a local machine, and q to denote an event parameter in a remote machine; where $p \in P$, and $q \in Q$ from our definition. In a synchronization we match parameter pairs in order of declaration; and from the semantics of decomposition/composition we say that $p = q$. We denote variables as v and w of local and remote machines respectively; where $v \in V$ and $w \in W$. In our current work we only consider variables of integer and Boolean type. In our implementations these map to Ada elementary types; parameter passing for elementary types, for *in* parameters, has call-by-value semantics; and for *out* parameters call-by-result semantics. With call-by-value, the actual parameter value is assigned to the formal parameter, $q := v$, on procedure entry. With call-by-result, the formal parameter value is assigned to the actual parameter, $v := q$, on return from the procedure.

In an abstract event we have assignment actions $v := w$; we will now show that our code implements this assignment, following translation using decomposed machines. During decomposition v and w are put into different machines. We introduce parameters, prior to decomposition, to facilitate this. In a refinement we introduce a parameter p to the development. Then the action $v := p$, with guard $p = w$, refines $v := w$. Decomposition then takes place; it gives rise to an AutoTask machine (local) event, with action $v := p$; this synchronizes with a Shared machine (remote) event with a skip action, and guard $q = w$. From event composition semantics we know that $p = q$. We now wish to show that our code implements $v := w$, when translated from the decomposed events. In our description of the code we use the same variable names, primed. So v translates to v' . We now introduce the notions of *formal/actual* and *in/out* parameters. We will say that the parameters of events translate to *in* or *out* parameters. This notion will be familiar to users of the Ada programming language. *In* parameters receive data from out of scope, and *out* parameters are used to send data out of scope. Since we can identify parameters by their location, i.e. parameters appearing in guards are sending (*out*) and parameters appearing in actions are receiving (*in*), we can talk about *in* and *out* parameters of events. Shared machine events always map to protected objects with protected procedures, so shared machine event parameters are always *formal*, and tasking machine event parameters are always mapped to *actual* parameters of a call, via parameters, so the parameters can be termed *actual*. This allows us to talk about *actualIn*, *actualOut*, *formalIn* and *formalOut* parameters. From Table 9.18 we know that parameter p in the AutoTask machine is an *actualIn* parameter, and in

the Shared machine q is a *formalOut* parameter. The translation gives rise to a procedure, in the protected object. The procedure body contains the statement $q' := w'$, where q' is a formal *out* parameter. The caller of the procedure supplies the actual parameter v' , so (following Ada's call-by-result semantics) on return from the call, v' is assigned the value of q' ; that is, $v' := q'$. We know that $q' := w'$, from the procedure body, and the return assignment $v' := q'$ follows the procedure body; therefore $v' := w'$ as required by the specification.

We now look at an assignment in an abstract synchronizing event, $w := v$; we show that our code implements this assignment, following translation using decomposed machines. During decomposition v and w are again put into different machines, and again, we introduce parameters, prior to decomposition. We introduce a parameter p to the development. Then the action $w := p$, with guard $p = v$, refines $w := v$. Decomposition gives rise to an AutoTask machine event with a *skip* action and guard $p = v$; this synchronizes with a Shared machine event, with action $w := q$. From Table 9.18 we know that parameter p in the AutoTask machine is an *actualOut* parameter, and in the Shared machine q is a *formalIn* parameter. The translation gives rise to a procedure, in the protected object. It has the statement $w' := q'$, where q' is a formal *in* parameter. The caller of the procedure supplies the actual parameter v' , so (following Ada's call-by-value semantics) on entry to the procedure, q' is assigned the value of v' . That is, $q' := v'$; and then $w' := q'$, in the procedure body; therefore $w' := v'$, as required.

Sensing and Actuating Events

In this subsection we describe how synchronized sensing and actuating events map to code implementing the Ada rendezvous mechanism; for communication between the controller and environment tasks. In the environment simulation, the sensing/actuating events map to an entry in the environment task, and a call in the controller task. The entry is an Ada specification, implemented by an accept clause. In the environment's accept clause, the monitored variable values can be assigned to the sensed variables of the controller task (sensing); or controlled variables in the environment task can be assigned the values of actuated variables of the controller task (actuating). An event is either actuating or sensing, but not both. Event Parameters are paired (using order of declaration) between the synchronized machines. In the case of sensing, we show how (using synchronization) the Event-B guards and actions are used to implement assignment of values in the environment to variables in the AutoTask Machine. In the case of actuation we show how they relate to the setting of variables in the environment. In the discussion

that follows we denote parameters as p and q , as in the previous subsection; and we denote variables as v and w . In this subsection we provide a slightly different interpretation, however, since variables v represent the sensed/actuated variables of the AutoTask machine (local). Variables w are the monitored/controlled variables in the environment (remote).

In a sensing implementation we want the environment to perform the assignment $v := w$. The value of the monitored variable w in the environment is assigned to the sensed variable v in the AutoTask machine. During decomposition v and w are put into different machines. We introduce parameters, prior to decomposition, to facilitate this. In a refinement we introduce a parameter p to the development. As a result of this, the action $v := p$, with guard $p = w$, refines $v := w$. Decomposition then takes place, and we obtain an event in the AutoTask machine, with an action $v := p$, and a synchronizing event in the Environ machine, with a guard $q = w$. Translation gives rise to an *accept* statement $q' := w'$, where q' is a formal *out* parameter. So (following Ada's call-by-result semantics) on return from the call, v' is assigned the value of q' ; that is, $v' := q'$. In the *accept* statement $q' := w'$ is followed by $v' := q'$. Therefore we have $v' := w'$, as required.

In an actuating implementation we want the environment to perform the assignment $w := v$. The controlled variable w in the environment is assigned the value of the actuated variable v in the AutoTask machine. In Ada, the statement $w' := q'$ is implemented in the body of an *environ* task *accept* statement. This is mapped from the Environ machine's synchronized action $w := q$. Here q is a *formalIn* parameter. On entry to the *accept* body, the formal *in* parameter q' is assigned the value of the actual parameter v' . So, $q' := v'$. From the assignment on entry to the *accept* body $q' := v'$ and then the *accept* bodies statement $w' := q'$, we can see that $w' := v'$, as required.

In Tasking Event-B sensing/actuating synchronizations are atomic, and this is enforced in the implementation by Ada's rendezvous mechanism in the implementation.

Addressed Variables

Tasking Event-B addressed variables are annotated machine variables of the Environ machine; or annotated event parameters (which are local variables) of AutoTask machine sensing, and actuating events. Parameters p of AutoTask machine events map to *local variables* p' in the implementation (they do not map to subroutine parameters). Machine variables w of an Environ machine, map to *variables* w' in the implementation. Each parameter p , of an AutoTask machine's *sensing* events, is paired with a variable w in the Environ machine. They share a common address *loc*. We annotate the

declaration of w as follows,

$$addr(b, loc) w ,$$

where b is the base, and loc is the memory location.

Now consider the case where a task senses the value of a monitored variable w from the environment. At the most abstract level, this involves making an assignment $v := w$, where v is a sensed variable in the task. Following decomposition we obtain synchronized events, with the action $v := p$ in the AutoTask event. In Tasking Event-B we annotate the parameter p with the location loc as follows,

$$addr(b, loc) p ,$$

where b is the base, and loc is the memory location. Therefore, the addresses of p and w are the same. The address declaration translates to Ada code as a variable declaration statement $p' : Integer$, followed by an address clause,

for p' Address **use** System'To_Address($b\#loc\#$).

Note the use of the ' character for accessing Ada attributes. In the *for* clause, p' Address, accesses the Address attribute of the variable p' . The Address attribute is set to the address object constructed in the *use* clause. The translation also generates a procedure declaration, in the task; from the AutoTask machine action $v := p$ we obtain the procedure body $v' := p'$. Remembering that p' refers to the value held at location loc , we have $p' = w'$. Then the assignment $v' := p'$ equates to $v' := w'$, as required by the specification.

We now consider actuating of a controlled variable w in the environment. At the most abstract level, this involves making an assignment $w := v$, where v is an actuated variable in the task. Following decomposition we obtain synchronized events, with the guard $p = v$ in the AutoTask event. In the Tasking Event-B we indicate that we want to read from loc by specifying an address annotation for parameter p , as above. The generated variable declaration of p' , and the address clause is also the same as the above. The translation generates a procedure declaration, in the task; from the AutoTask machine guard $p = v$, we obtain the procedure body $p' := v'$. Remembering that p' refers to the value held at location loc , we have $p' = w'$. Then the assignment $p' := v'$ equates to $w' := v'$, as required by the specification.

– A Note about atomicity.

In a specification, the synchronizing events atomically set, and read from, variables in the environment. So, an event that writes to two controlled variables in the environment $w1 := v1 \parallel w2 := v2$ has a guarantee that

if $v1 = v2$ then $w1 = w2$. In the approaches presented up until now the protected procedure and task entries ensure that this is the case. However in the implementation accessing memory directly it is implemented in a serial fashion $w1' := v1'; w2 := v2$. In this case there is no guarantee that $w1 = w2$ if $v1 = v2$. We therefore need to restrict sensing and actuating to a single action per event to maintain the refinement relation.

9.1.8 Generating and Executing Code

Code is generated from the Tasking Event-B model in a two-step process. The first step generates a common language model (CLM), from which a number of target implementations could be generated. The CLM is an abstraction of commonly used software constructs. Translation to a specific target language takes place in the second step. Our current tool generates a pretty print of Ada code, which we copied and placed in appropriate files in an Ada development environment. We successfully compiled and ran the code with no further changes to the automatically generated code. The environment simulation, that we specified, manipulated changes to the monitored variables; changes were reflected in the environment's controlled variables after the heater controller tasks had reacted. The variable values in the environment are written to a console, using the *Output* construct, to provide feedback.

9.1.9 Conclusions

The main focus of the work that we report here is modelling of, and providing implementations for, multi-tasking embedded control systems. We initially model the system using Event-B, but to generate implementations in the coding style that we desire, we have the need for features that do not exist in Event-B. We add various modelling constructs, in an extension to Event-B, that are used to model implementation-level details, including the controller's interaction with the environment. We make use of the extensions to guide code generation. Our contribution is a methodology, and tool support, for facilitating this. We have validated the method and tools using a case study, which provides a basis for future developments in our sphere of interest.

Discussion

In our previous work [Edm10, EB08, EB10], we found that models quickly became intractable; the models became very large and generated a large

number of proof obligations. To address this problem we use the decomposition approach of [But09a, SPHB10]. After decomposition, the models can be refined further, and when we are ready to provide implementation details, we use the constructs introduced in the extension. The annotations and Event-B models are then used to generate code. In previous work we described an object-oriented intermediate language that we used, to guide code generation. However, we encountered difficulties due to the large semantic gap between Event-B and the intermediate specification language. In the work presented here, the methodology maintains a small semantic gap, we achieve this by adding only a minimal number of constructs to the Event-B language. The implementation of a machine, for instance, can be either a controller task, environment task, or protected object. We apply a single annotation to the machine, which describes the implementation. In the case of the tasks, we add a task body annotation describing the event ordering; which includes branching and iteration constructs. Individual events can map to a number programming constructs; their implementation is defined by their use in the task body, and by any additional annotations.

In previous work, the restriction on inter-task communication forced us to adopt a particular modelling style, and this did not reflect the way that systems are implemented in industry. We removed the restriction on inter-task communication, making the the environment machine a special case. We introduced new event annotations, sensing and actuating; and a new parameter annotation, addressed variables. These new features use the existing synchronization approach, arising from model decomposition, to model interaction with the environment. The annotations guide the translator to produce the appropriate code. The work presented here, also includes a solution for modelling low-level interaction with the environment; that is, sensing and actuating. We take into consideration that, in the final deployable system, inter-task communication is prohibited. The main driver for this restriction is that we wish to generate safe multi-tasking code which is compliant with the Ravenscar subset of Ada [BDV04]. However, we are able to relax this restriction for environment tasks, since our primary task is to simulate the environment. The sensing and actuating events give rise to two styles of implementation: one using entry calls, and one using addressed variables. In the entry-call style, calls are a way of updating sensed variables in a controller task, and controlled variables in the environment task. The entry is implemented in the environment, and called from the controller task. In the Ada translator we generate entries in the environment task specification, and these are implemented by **accept** clauses in the task body. We may find, in future work, that this approach has a role to play when implementing the interaction between tasks and external driver APIs. The second

style, addressed variables, associates address information with each of the parameters of a controller task's sensing or actuating events. In our case study, a parameter with address and base, is translated to a variable which is declared using Ada representation address clauses. Other languages, like C, would reference the address using a pointer. When the variable is used in an expression, the specified memory address is read from, or written to.

The case study of subsection 9.1.4 was proposed in the course our Deploy project [EU-]. The models arising from this activity are available at [ERa], and form part of a tutorial available at [ERb]. It is here that one can find a listing of all the generated code.

The case study was used as to validate the approach. We have shown that it is possible to use Tasking Event-B to model a system, and its environment, in such a way that leads to generated code of use to industry. The annotations facilitate automatic generation of deployable controller code, and a simulation of the environment. The partitioning of the tasks in the controller and the environment, using Event-B decomposition, is quite intuitive. The semantics of event synchronization allow us to model the interaction between the decomposed tasks of the controller and environment.

There are a number of avenues that may be explored in future work. For instance multiple Environ machines could be used; that is one machine for each device in the environment. The task entries may provide the basis for a link with device driver APIs. It would also be useful to explore interaction with a modelling tool such as Simulink.

Related Work

The closest comparable work is that of Classical-B's code generation approach [Abr96] using B0 [Cleb]. B0 consists of concrete programming constructs, these map to programming constructs in target programming languages. B0 can be translated to Ada, but there is no support for concurrency. The Event-B and Classical-B approaches differ in many respects. For example, the Event-B approach supports modelling of the controller, and environment together; Classical-B is aimed more at modelling software systems in a modular fashion, and the generated code is sequential, and does not consider concurrency. In the work presented here, we have added the ability to provide an Ada implementation of the deployable code, and an environment simulation. Code generation of B to embedded systems was carried out in [BBP⁺03], where the implementation results in sequential code. Some consideration is given, in [SCZ07], to the use of an Event-B-like syntax for analysis of multi-tasking programs. In comparison, we use the task body for scheduling, rather than taking a purely interrupt driven approach; we have

yet to incorporate modelling of interrupts in Tasking Event-B.

VDM++ [CSK] may be used to generate code, it is an object-oriented extension to VDM-SL formal specification language. It has been used to model real-time systems, see [VLH06]. The paper describes a controller and environment model similar to our own. They go on to define an abstract operational semantics, which is quite general, and has additional features when compared to our work. They model time, and asynchronous communication. We do not address these issues in our work since the research is ongoing.

Scade [Ber07] is an industrial tool for formally modelling embedded systems. It provides a graphical approach to specification, and has a certified code generator. It has a similar control flow approach to that of UML-B statemachines [SB08]. However, we have not investigated code generation from UML-B to Ada.

In [GSLB03] Karsai et al. describe some common elements of modelling approaches for embedded software development. The work that we present here is comparable in some ways, since we can view our approach as an example of the Model-Integrated Computing [NK04] that they describe. Their approach suggests development is based on models and generation, which ours is; and that the approaches have multiple views which ours has (for example UML-B [SB08]). They also advocate an extensible approach: the Rodin tool has a built-in extension mechanism, and our tools are based on a extensible meta-models. The approach differs in that they focus on hardware architecture and signal-flow aspects of the design, and use a Finite State Automata [Hen96] approach to formalize the dynamic aspects of the system. Event-B is a state-based approach, but does have analysis such as ProB [LB08] to address dynamic aspects.

9.2 Code Generation and Scheduling of Event-B Models

In this section, we suggest an approach to code generation in Event-B and scheduling of the resulting code. We have written a plug-in for the RODIN platform that translates models into C++ [Str00] code, which can then be compiled into object code. The generated code consists of C++ methods, which can be executed in parallel using a separate scheduler that we have developed. Scheduling depends on a behavioural semantics, which for our tool is inherited from the Action Systems formalism [BKS83], and which allows for a parallel interpretation of programs. We support parallel execution of independent events by using the MPI (Message Passing Interface) framework

[Mes], which allows events to be distributed over several cores or processors, as well as over a network. The work presented in this section is based on the master’s thesis of Grönblom [Grö09] as well as on a technical report [DGS11].

The rest of the section is structured as follows. In 9.2.1, we focus on our code generation plug-in that can translate a certain class of Event-B models into C++ code. Next, in Section 9.2.2, we present a scheduling tool designed to execute the code produced by the plug-in on a topology consisting of one master and several slave nodes. We also discuss scheduling policies as well as parameters that can be given to control what events are executed on which node. We then give a practical example in Section 9.2.3, in which our approach is used to generate code from a factorisation model and execute it on processors in a cluster. Finally, we sum the material up in Section 9.2.4, where we also discuss related work.

9.2.1 Code generation

Event-B in its basic form does not support translation of models into executable code. We now show a means of translating a model into object code that can be executed as a computer program. It is based on a method proposed by Wright [Wri08], which was originally intended for use with a virtual machine framework. In our approach, translation is made from the last refinement step, also called the ultimate refinement of a model. Other attempts at generating code from Event-B models are summarised in Section 9.2.4, where we compare them to our framework.

Approach summary

To achieve code generation, we have developed a plug-in for the RODIN platform. The plug-in accepts a certain subset of Event-B models, and translates them into C++ code, which is based on the C language, but possesses additional high-level features. However, C++ specific features are mostly not used by the code generator, and it could easily be modified to generate standard C code instead. Since both C and C++ are widely used and their compiled code is generally fast, we consider both of them to be suitable as a target language for model translation, which is important in parallel computing.

The code generator is written in Java as a plug-in for the RODIN platform. The Eclipse IDE, which RODIN’s interface is based on, provides good support for plug-in development. It provides an application programming interface (API) for creating the user interface of a plug-in. A plug-in can access

models that have been created with RODIN, from a database. Event-B components are fetched through an interface to the database. For example, one can fetch the invariants of a machine through a function call. This interface is used by the code generator to fetch the components of a model.

Our code generator does not only translate code on a one-to-one basis, but also adds components needed for parallel execution in the scheduler, such as the dependency matrices that we discuss in Section 9.2.2. Since such computations would be cumbersome to do by hand, we consider tool support mandatory for our framework. The output of the code generator consists of a model converted to a C++ class, in which the events of the model are represented as methods. The idea is that our scheduler (discussed in Section 9.2.2) executes and accesses the class through an *interface* that we have defined. An interface consists of a set of operations that can be executed by the clients. The interface of the model contains several different operations on the model. For example, it contains operations for executing an event, checking a guard and accessing a variable.

Event-B0

In most cases, an Event-B model cannot be elementarily translated into computer code of some programming language. This is because specification languages are very different from programming languages. They both serve a different purpose and there is no one-to-one mapping between them. In our opinion, the best way to handle this difference is to define a subset of Event-B that consists of components having a direct equivalent in the target language. We have defined such a “concrete” subset, called Event-B0. This decision was inspired by a very similar set, called B0, defined in classical B [Clea], which contains a subset of classical B with some additional constructs. B0 contains only concrete data types and operations on these, but not any abstract components that would be non-trivial to translate. It is equivalent to a limited programming language that can be converted into programming languages such as Ada or C. Components included in B0 are, among others, *integers*, *arrays*, *enumerated sets* and *arithmetic operations*. Atelier B and the B Toolkit both include a code generator that converts B0 models into computer code. For this process, both tools have a separate B0 checker that checks if a model can be translated.

Ranges of variables. The components that we have allowed in Event-B0 are those that have a low level equivalent in C++. They are very general and can be expressed in most programming languages. Another restricting factor of Event-B0 is the fact that computer memory is limited, so the size of

every type and data set has to be assigned a limited size. Otherwise, it would be possible for variables to *overflow*, which is a situation when a variable is assigned a value outside its available storage space on the computer. These situations normally lead to incorrect behaviour. Integer overflow in the C programming language causes undefined behaviour [KR78], so we have to prevent situations where it can occur.

Most programming languages have minimum and maximum integer values, often denoted `INT_MIN` and `INT_MAX`, respectively. Our target computer uses signed 32-bit integers with a range between -2^{32} and $2^{32} - 1$, from -2147483648 to 2147483647, in C++ programs. If the value of a variable goes beyond this range, the program will not be valid. The size of enumerated sets and arrays can neither be of infinite size. A maximum size has to be defined in an invariant.

To prevent overflowing, we impose restrictions on the arithmetical operations in Event-B0. We only allow operations that have two factors. If there are more than two factors, the whole expression can be valid, even though a part of the expression overflows. For instance, the arithmetic operation $(\text{INT\_MAX}+1)-1$ overflows in the first part. The maximum integer value plus one cannot be represented by the storage space of the integer. However, the whole expression is of legal range, as the result is `INT_MAX`.

However, there are techniques for handling integer overflow. It is common that *wrap around* is used in situations of this type. If an arithmetic expression overflows, it continues the operation from the opposite extreme. This is similar to how the modulo operator works. For example, if an arithmetic addition overflows with 5, the result will be evaluated to `INT_MIN+4`. By using wrap around, the operation $(\text{INT\_MAX}+1)-1$ would be evaluated correctly on a computer. First, $(\text{INT\_MAX}+1)$ would overflow so the value will be in the opposite extreme, i.e. `INT_MIN`. The second part of the expression, `INT_MIN-1`, will also overflow and be evaluated in `INT_MAX`. Hence, the correct result is achieved. Arithmetic operations in Event-B0 would be less strict by using a target language that supports wrap around.

Invariants. Invariants are used in Event-B0 for assigning types and restrictions to variables. Every variable has to be assigned a type of either integer, Boolean, enumerated set or an array of one of these types. Event-B0 has no other restrictions on the invariant, as they only concern the verification part of the model, but not the functionality. Therefore, they will not be needed in the executable version, and do not have to be translated.

Formal presentation of Event-B0

We now present Event-B0 formally by using simplified *production rules* in *Backus-Naur Form*. Variables are defined in an identifier and assigned a type in an invariant. The type is assigned by using the “belongs to” operator, denoted by the symbol $\in$. Arrays are defined using the function operator, denoted “ $\rightarrow$,” from a numerical interval, 0 to n , that maps to one of the three Event-B0 types. Type assignments are defined in the following way:

TypeAssignment ::=

$Identifier \in (BOOL|integer|enumerated\ set)$

ArrayTypeAssignment ::=

$Identifier \in 0..N \rightarrow (BOOL|integer|enumerated\ set)$

Variable assignments have to be written in a very simple form in Event-B0. On the left-hand side we have a single variable or array element and on the right-hand side a variable, a value or a two-part arithmetic operation. One important thing to note is that events are atomic, which means that all the substitutions of an event are executed at once. This has the effect that variable assignments are updated only after the whole event has been executed. This is not the case in C++, where any variable assignments instantly take effect. Hence, we have to forbid events that first update a variable and then use it in a later substitution, as it would potentially lead to incorrect C++ code. Assignments are formally expressed as follows:

Assignment ::=

$Identifier := (Identifier|Value|ArithmeticOperation|BOOL)$

The arithmetic operations allowed in Event-B are the four basic ones: *addition*, *subtraction*, *multiplication* and *division*, including the *modulo* operation. They can have one value or one identifier on each side:

ArithmeticOperation ::=

$(Value|Identifier) (+ | - | * | / | \text{mod}) (Value|Identifier)$

Event guards are predicates that have to be true for the statements of the event to be executed, i.e. event preconditions. Event parameters are defined in the same section as the guards. In Event-B, parameters can be assigned a type implicitly. For instance, a parameter occurring in an arithmetic predicate will be assigned a numerical type automatically. This is similar to implicit type casting in dynamic languages. However, in Event-B0, every

parameter has to be assigned a type explicitly, as in static programming languages. Guards can be expressed as follows:

RelationalExpression ::=
Expression RelationalOperator Expression
Expression ::=
(Identifier | Value | ArithmeticOperation)

The relational operators can be of six different types in Event-B0. Integers can be used with all operators. Booleans and enumerated sets can only be used with the “equals” and “not equals” operators. The following relational operators are allowed:

RelationalOperator ::=
 (= | ≠ | < | ≤ | > | ≥)

Operation of the code generator

The plug-in starts by fetching the most refined machine in a chosen Event-B model, constituting the most concrete version in the refinement chain. This machine needs to be in Event-B0 form in order to be translatable by the code generator. One problem that we encountered with the refinement aspect of Event-B is that some elements can only be found in abstract machines from earlier steps of the refinement chain. With components scattered across several machines and contexts, one would have to first merge all machines into one concrete machine. However, this is not supported by our code generator, as we have not focused on model merging. Therefore, the code generator demands that all components that are used in the most refined machine be situated in it. The code generator performs translation and Event-B0 checking simultaneously. It checks on the fly if components are of Event-B0 form and if they are, it proceeds to convert them to C++ code. If an illegal component is encountered, the plug-in terminates with an error code describing why the component could not be translated.

Every event of a machine is translated into two separate functions: one that contains the guard and one that contains the assignments. This design decision was taken due to the fact that the guard checking is done by the master node before an event is run, and the slaves only execute the substitutions of the event.

Event-B	C++
MACHINE m VARIABLES <i>var1</i> <i>var2</i> <i>var3</i> <i>var4</i> INVARIANTS <i>var1</i> $\in$ -2147483648..2147483647 <i>var2</i> $\in$ <i>BOOL</i> <i>var3</i> $\in$ 0.. <i>N</i> $\rightarrow$ -2147483648..2147483647 <i>var4</i> $\in$ 0.. <i>N</i> $\rightarrow$ <i>BOOL</i>	int var1 bool var2 int var3[N] bool var4[N]

Figure 9.19: Translation of type assignments.

Translation rules of the code generator

This section briefly discusses how the components of an Event-B model are translated into C++ code. In Event-B machines, program variables are defined in two steps. The variable first has to be defined by a unique name in the VARIABLES clause. The variable is then assigned a type in an invariant. It is also possible to add other restricting invariants to variables. However, such invariants will not be converted to code because they concern the correctness of the model, not the functionality. Translation rules for type definitions are defined in figure 9.19.

In every Event-B model, all variables have to be assigned initial values in the *INITIALISATION* event. Integers, Booleans and enumerated sets always have to be given a deterministic value. However, we have not implemented a way of initializing arrays, as large arrays would require hundreds of substitutions in the initialisation. The code will automatically initialise the elements to values that the compiler has chosen. However, for correctness, RODIN demands that all variables be initialised. Arrays have to be initialised to some “dummy” values in a non-deterministic assignment. It would be possible to add some mechanism for initialising arrays that can be converted to C++ code.

Event-B	C++
CONTEXT c	<code>const int const1 = value</code>
CONSTANTS	
<i>const1</i>	
AXIOMS	
<i>const1 = value</i>	

Figure 9.20: Translation of constants.

Constants can be defined in contexts, by an identifier in the **CONSTANTS** field and a value assignment in an axiom. Constants are useful in parallel programming as they can be accessed simultaneously by several processes. Constants are translated into code as described in Figure 9.20. Detailed descriptions of how other Event-B constructs are translated are given in [DGS11].

If a model is in Event-B0 form, then the code generator will translate it into two different C++ files that the scheduler can execute. The machine and the contexts are translated into the header file “Machine.h” and the source code file “Machine.cpp.” The C++ class that represents the model is defined in the source code file. The interface to the model is also located in this file. Enumerated sets and constants that are defined in a context are situated in the header file. The code generator also generates two files for handling parameters: the header, “Environment.h,” and the source code in the “Environment.cpp” file. Custom functions that create parameter values can be added to the latter file.

9.2.2 Scheduling

In order to execute the code generated by the plug-in, we have also developed a scheduling platform. This tool has its roots in a method given by Degerlund et al. [DWS07] of correctly scheduling an action system in a parallel environment. Scheduling in this sense means assigning computational work to processes. As the Action Systems formalism targets parallel and distributed systems, its structure is readily suitable for parallel execution. The only rule that has to be followed is that events that have no variables in common can be executed in parallel. To execute an action system in parallel, we have to fulfil this criterion for correct behaviour. Degerlund et al. achieve

this by applying mutual exclusion to the variables, so that they cannot be accessed by two or more events simultaneously. Events that need to access a variable that is currently in use by another event have to wait until it is freed up. Note that the terminology used in action systems theory is somewhat different from that of Event-B. For example, the word *action* has a different meaning in action systems (corresponding to *events* in Event-B) as compared to how it is used in Event-B. For clarity, we will stick to Event-B terminology, even when discussing theory with its roots in the Action Systems formalism.

Scheduling tool support. Degerlund et al. developed the proof-of-concept program *ELSA* that implements the described scheduling method for classical B models. As most of the research in the field has been theoretical, focus was put on implementation issues. ELSA can schedule classical B models of B0 form that have been translated to computer code. Models are created with the Atelier B tool and then converted to C code by the built-in code generator. ELSA is written in C++ and utilises the MPICH2 communication library [MPI] for communication between nodes in a computer cluster. MPICH2 is an implementation of the Message Passing Interface (MPI) [Mes], which is the most dominant communication protocol used for parallel programming [GKP09]. The MPI standard has been widely used in computer clusters and supercomputers. MPICH2 provides an interface for communication between computers by *message passing*.

The Event-B scheduler is based on the code of ELSA, but contains changes and added functionality needed for Event-B and our framework. The scheduler is written in C++ code and utilises the MPICH2 communication library for parallel computing. This set-up was considered to be suitable for our requirements. The scheduler takes as input an Event-B0 model, which has been translated to C++ code by the code generator. Upon execution, it schedules the events in parallel on a computer cluster or a computer with multiple computational cores or processors.

Background theory

The scheduler is based on the parallel interpretation of actions systems / Event-B stating that events that share no variables can be executed simultaneously. To schedule an event, two properties must hold:

1. The guard of the event is true.
2. No other events that share variables are currently being executed.

To avoid interference, we utilise mutual exclusion to prevent variables from being accessed simultaneously by several events. This is implemented by using a locking mechanism. An event in execution locks all variables involved in the event, so that no other events can access them. The variable locks are implemented by Boolean variables. If a variable is locked, then a corresponding Boolean locking variable will have the value true, and if the variable is not in use, the lock will instead have the value false. We only need to implement this mechanism on the master node, as it takes all scheduling decisions. The locks are implemented using a matrix that is created in two steps.

Variable-event matrices. When executing an action system or an Event-B model in parallel, we have to prevent several events that share variables from being executed simultaneously. To achieve this, we first generate the *variable-event* matrix, ve , of the size $n * m$, where n is the number of global variables and m is the number of events in the system. For every element ve_{ij} , where $i \in \{1..n\}$ and $j \in \{1..m\}$, in the matrix, we will have a Boolean value. If variable i is involved in event j , the element will have the value true and if it is not involved, it will have the value false. By “involved,” we denote that an event accesses a variable. This matrix describes the dependencies between the events and the variables.

Event-event matrices. After the variable-event matrix has been created, we can create a scheme of the dependencies between events, with respect to the variables. This is described in the *event-event* matrix, ee , of size $m * m$, where m is the number of events in the system. This matrix also contains Boolean values and is derived from the ve matrix. Element ee_{ij} is true if there exists a $k \in \{1..n\}$, where $ve_{ki} = ve_{kj} = \text{true}$, otherwise it will have the value false. This means that if the events i and j both use variable k , they have a variable dependency and cannot be executed simultaneously.

Event-location matrices. We also have a third matrix, el (*event-location*), for specifying which events are allowed to run on the processors. An action system can be partitioned with respect to either variables or events [Ser90], dividing them into disjoint sets. By partitioning a system with respect to events, every processor has a set of events that can be executed on it. This scheme is represented by the el matrix of the size $m * p$, where m is the number of events in the action system and p is the amount of processors used on a computer cluster. If element el_{ij} , where $i \in \{1..m\}$ and $j \in \{1..p\}$, is true, then event i is allowed to be executed on processor j . If el_{ij} is false, then it is not allowed to be executed on processor j . Every processor should

have at least one event that is allowed to be executed on it, and each event should be executable on at least one processor. This matrix can, for example, be used to reserve a faster processor for a compute-intensive event.

Scheduler considerations. We have implemented the three above-mentioned matrices for our Event-B scheduler. The code generator calculates the two dependency matrices upon model translation. The *el* matrix can be defined in a configuration file of the scheduler. This matrix is optional for the scheduler. If it is not defined, all events can be executed on any processor.

The scheduler tries to schedule events to the nodes in the cluster in a *round-robin* fashion, as the execution order of an Event-B model is non-deterministic. It first checks that the guard of an event is true and that it is not currently being executed. Then it checks whether the event shares any variables with other events currently being executed. If this is the case, the event cannot be scheduled. Finally, it checks if there are any idle nodes that can execute the event. After a slave node has executed an event, it returns the updated values of all variables involved in the event to the master node, after which the variables can once again be accessed by other events.

When all guards of a machine are false, the scheduler is considered to be *deadlocked*. This is often a desired termination state. However, some systems will always have a guard evaluating to true and can therefore execute infinitely. This is often the case in reactive systems that are designed to execute forever. A system of this type has to be terminated manually. An Event-B machine is said to be deterministic if only one guard is true at any time. Such machines will not gain any speed from parallel execution, as only one event can be executed at once.

Fairness of events concerns the order of event execution. It is possible that an event can be blocked by other events in execution, either because of mutual exclusion or because all processors are constantly busy. In the former case, we have implemented some fairness, by always checking the guards in different order. Every time the scheduler iterates through the guards, it starts from a different one. Otherwise, an event could constantly disable other events by always being executed first. It is also possible to exercise manual control by editing the *el* matrix. For example, a specific processor can be reserved for a given event.

Operation of Scheduler

In our parallel execution of Event-B models, we have a central master process that schedules events to be executed on slave nodes, as illustrated in figure 9.21. The master node has the most up-to-date state of the Event-B model

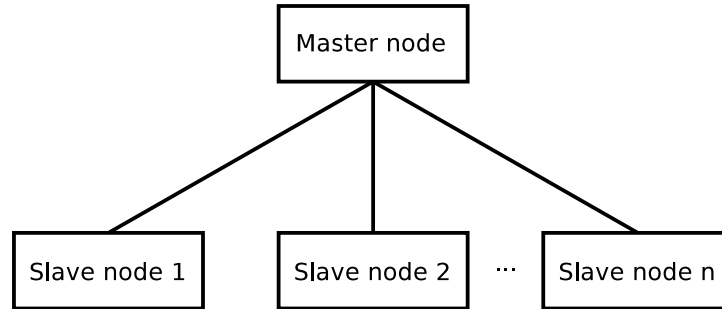


Figure 9.21: Relationship between master and slave nodes.

and it takes all scheduling decisions. Upon scheduling an event to a slave node, the master node sends all the variables involved in the event to it. The slave proceeds to execute the event and once finished, it sends back all the variables to the master node. Then the master updates the master state of the model and frees up the variables that were involved in the event.

Scheduler algorithms

The master node constantly schedules events to the nodes in a loop. It schedules any event that satisfies all the conditions described above. The main scheduling algorithm is expressed in pseudocode in figure 9.22.

After the scheduler has looped through all events once, it waits for a slave node that has finished execution to send back the results. If no events are being executed, then the program terminates. This algorithm is described in pseudocode in figure 9.23.

The slave nodes have a different algorithm that they constantly loop through. All the slave nodes continuously wait to be scheduled an event. After receiving an index of an event to execute, they receive the values of the variables involved in the event and any parameters it might have. It then proceeds to execute the event by using the variables it received. After finishing, it returns the updated variables to the master node and idles until it is scheduled another event. This algorithm is described in figure 9.24.

9.2.3 Example: Factorisation

We now show how our method can be used in practice using an integer factorization example. The scenario is that we have an integer n , of which we want to find the lowest factor (greater than 1). If n is a prime number, the result produced should be n itself. The algorithm we use is trial division, which is in itself ineffective, but simple enough to show how our method

```

WHILE any guard true
    DO iterate through events 1..n with i
    IF event i is already running
        THEN stop current iteration
    IF variables involved in event i in use
        THEN stop current iteration
    IF the guard event i is false
        THEN stop current iteration
    IF event i has parameters
        THEN randomise parameters
    IF idle nodes available
        THEN schedule event i to free node
    send variables involved in event i to the node
    IF event i has parameters
        THEN send parameters to the node

```

Figure 9.22: Master node scheduling algorithm.

```

IF any events are currently in execution
    THEN
        wait until any node has finished execution
        get index of executed event
        receive variables involved in the event
        free up variables involved in the event
    ELSE send termination signal to all nodes

```

Figure 9.23: Master node receive results algorithm.

```

DO
    receive an index of scheduled event
    IF index equals termination signal
        THEN terminate
    receive the variables involved in the event
    IF event has parameters
        THEN receive parameters
    execute event
    signal master node that execution has finished
    send back the variables involved in the event

```

Figure 9.24: Slave node algorithm.

works. The problem being *embarrassingly parallel*, it is also easy to share the tasks between different processors. The factorisation algorithm is performed by checking whether $n \bmod i$ is zero, for values of i starting from 2 counting up to $\sqrt{n}$. If a factor has not been found by then, it can be shown mathematically that n is a prime number. Since Event-B does not support square root, we instead use $n/2$ as our limit, which is correct, albeit not as efficient. As soon as a match is found, the algorithm terminates and the corresponding value of i is returned.

The parallel version of the algorithm works in the same way, except that it performs several trial divisions concurrently. Every process has a different set of divisors, with the first one performing the modulo operation with i and the second one with $i + 1$, the third one with $i + 2$ etc. Instead of increasing the i variable by one, we increase it with the process amount. In this way, every event uses a different set of values for i . By defining n as a constant in the context, it can be used by all events. Race conditions can not occur when accessing constant values concurrently. The algorithm would normally be performed by having the same function executed on every node, but with different parameters. Such algorithms cannot be modelled in Event-B, so instead, several different events that perform the same computations can be created. One drawback with this is that a dynamic number of threads cannot be used to execute the algorithm.

Test case modelling. In our test case, we have modelled parallel trial division with three different events that perform modulo operations simultaneously. For every event, there are three associated variables: an $i_{process}$ variable that is used in the modulo operation, a $result_{process}$ variable for storing the result of the modulo operation, and a Boolean variable called $continue_{process}$ for controlling program execution. The $process$ subscript denotes a process number, which in our model is 1, 2 or 3. The variable $i_{process}$ is increased by 3 after every trial division, so that all events have a different set of values for $i_{process}$. The $continue_{process}$ variable has to be true for the trial division to be carried out, and after execution, it is set to false by the event. A fourth event, called *check*, sets all the *continue* Booleans to true if no divisor to n has been found, and if there is still a possibility to find one. The program continues until the smallest divisor to n has been found, or until each counter $i_{process}$ has exceeded $n/2$. In the latter case, n is a prime, as it has no other factors other than 1 and itself.

Results. We executed the model successfully, with different values for n , on a computer with eight cores. The algorithm was modelled as one machine and one context, both in Event-B0 form. The model was automatically translated to C++ code by our code generator. By computing and examining the *event-event* matrix of the code, we can see that events 1 to 3, the trial division events, share no variables. Hence, they can be executed in parallel. Event number 4 is the event that checks the result. It is involved with the variables of events 1 to 3 and can therefore not be scheduled while any other events are running. Machine and context listings can be found in [DGS11].

9.2.4 Conclusions and future work

In this section, we have discussed an approach to code generation using the Event-B formalism, as well as scheduling and execution of the resulting code. Software can be modelled and refined in the established Event-B tool (the RODIN platform) in the standard way, but the last refinement step has to comply with Event-B0, which is a subset of the Event-B language. Event-B0 is inspired by the B0 language of the Atelier B tool used for the classical B method, and it only contains constructs that can be easily translated into C++. While the correspondence between the two languages has not been formally proven, Event-B0 has been carefully designed to contain only constructs that have a very close correspondence to C++ code. Events are translated to C++ methods, and the constructs allowed in the events are restricted. Translation from Event-B0 to C++ has been implemented as a plug-in for the RODIN platform.

We have also developed a scheduler, written in C++, that is used to execute the generated events (methods). Execution adheres to the commonly used behavioural semantics of Event-B, in which enabled events are non-deterministically chosen for execution, and the program terminates when all events are disabled. The scheduler makes use of the fact that events are assumed to be atomic, and it is therefore possible to schedule events in parallel, given that all events executed in parallel have no variables in common. The scheduler uses the MPI (Message Passing Interface) framework to schedule events on different processors in a network, or on different cores of a multi-core processor.

Related and future work. Code generation for Event-B and related formalisms has also been studied elsewhere. The Atelier B tool for the classical B method defines a B0 language [Clea], from which its code generator is able to produce C/C++ or Ada code. The intermediary B0 language serves the corresponding purpose as Event-B0 in our work, and Event-B0 is in fact inspired by B0. Atelier B does not take a stand on a behavioural semantics, and the operations (cf. events in Event-B) are simply translated into functions in the target language, but have to be explicitly called upon by the programmer. Consequently, Atelier B does not provide a scheduler, nor does it explicitly take a stand on concurrent execution. A scheduler that can be used with (slightly modified) code generated from Atelier B has, however, been developed by Degerlund et al. [DWS07]. This scheduler is intended for use in B Action Systems, and it also constitutes the code base for the scheduler discussed in this section.

A code generator for Event-B similar to ours has been developed by Wright [Wri08]. It was, however, developed for the purpose of a virtual machine project, and was not intended to be an all-round tool. Our code generator can be seen as an extension, taking Wright's work one step further towards a general tool. A similar tool, EB2ALL, with C, C++, C# and Java code generation has been developed at Loria [Lor]. To our knowledge, it focuses strictly on code generation, and does not take a stand on scheduling. Edmunds has suggested an Event-B code generation approach [Edm10] in which the developer can express control flow information in a language called OCB (Object-oriented Concurrent-B). The target language is Java, and concurrent execution is supported. This method gives the developer more control over the execution of the final program, which is sometimes a desired feature. It is, however, a different design philosophy as compared to ours, where we rely on the established Event-B behavioural semantics and let the scheduler automatically take care of the scheduling in such a manner

that adheres to the semantics. This approach has also been further developed by Edmunds and Butler and adapted to ADA code generation [EB11]. Another approach to Event-B scheduling has been proposed by Boström [Bos10]. Boström’s work, which focuses on sequential programs, relies on explicit schedules given in a scheduling language, and proposes a pattern-based approach to showing the correctness of imposing a given schedule on an Event-B model. Related methods have been suggested by Iliasov [Ili09] and Hallerstedte [Hal10]. An approach that introduces support for concurrent programs has also been suggested by Boström et al. [BDSW11].

The automated “on-the-fly” scheduling of our approach has the advantage that it very closely preserves the usual behavioural semantics of the models. It also facilitates for the developer, since the execution order of the events is automatically decided by the scheduler during run-time. On the other hand, the lack of explicit control flow also poses a challenge. Since the events are scheduled one by one on different processors/cores, the communication overhead is sometimes large. Future work includes evaluating how this affects the practical use of our method, and also investigating means of (automatically) scheduling groups of events, or repetitive execution without involving the scheduler. The challenge constitutes achieving this in a way that adheres to the standard behavioural semantics and, thus, would not require the introduction of explicit control flow structures.

Chapter 10

Timing and Probabilities

*J. Bryans, M. Butler, J. Fitzgerald, S. Hallerstede, T. Hoang, L. Laibinis,
I. Lopatkin, M. Sarshogh, K. Sere, A. Tarasyuk, E. Troubitsyna,
L. Tsiopoulos*

Clearly timing is important in many embedded systems applications. While Event-B does not have explicit support for timing, it is possible to model discrete timing properties by introducing a discrete clock into models. This chapter presents an approach to modelling discrete timing by using standard patterns of timing properties and their refinement.

Probabilistic reasoning is important for analysing the reliability of safety-critical systems and this chapter also describes approaches to extending Event-B to treat probability. A first approach considers cyclic systems and shows that their behaviour can be represented via a common Event-B modelling pattern. We show then how to augment such models with probabilities (using a proposed probabilistic choice operator) that in turn would allow us to assess their reliability.

Related to this, we describe an approach that uses design strategies that allow the developers to structure safety requirements according to the system abstraction layers. Essentially, such an approach can be seen as a process of extracting a fault tree – a logical representation of a hazardous situation in terms of the primitives used at different layers of abstraction. Eventually, we arrive at the representation of a hazardous situation in terms of failures of basic system components. After augmenting Event-B specifications with an explicit representation of probabilities of component failures [TTL10], we can use standard calculations to obtain a probabilistic evaluation of a hazard occurrence. As a result, we obtain an algebraic representation of probability of safety requirements violation. This probability is defined using the probabilities of system component failures. To

illustrate our approach, we present a formal development and safety analysis of a radio-based railway crossing.

10.1 Modelling Timing Properties In Event-B

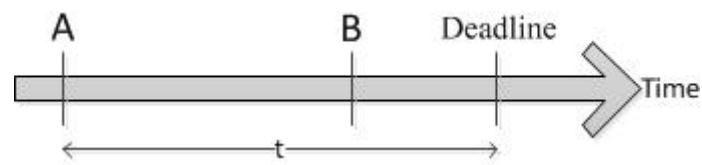
In this Section, first three groups of timing properties which are the main focus of this research will be introduced. Then our approach to formulate the process of encoding them in Event-B will be discussed in detail. Our formulation approach consists of some annotations in order to express timing properties and translation of those annotations to invariants, guards and actions. In the end, since refinement is one of the most important features of Event-B, some patterns to refine abstract timing properties to concrete ones based on control flow refinement, will be introduced.

10.1.1 Time Properties Categories

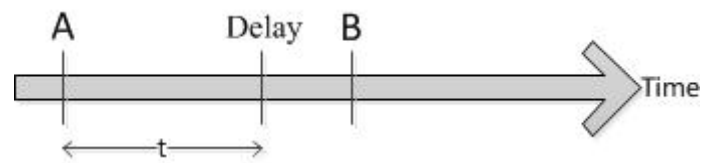
In order to explicitly represent timing properties we extend the Event-B syntax with constructs for deadlines, delays and expiries. Each of these timing properties places a discrete timing constraint between trigger events and their response events. A typical pattern is a trigger followed by one of several possible responses, thus each of our timing constructs specifies a constraint between a trigger event A and a set of response events $B_1..B_n$. The syntax for each of these constructs is as follows:

- **Deadline**($A, \{B_1, \dots, B_n\}, t$)
(Figure 10.1a),
- **Delay**($A, \{B_1, \dots, B_n\}, t$)
(Figure 10.1b),
- **Expiry**($A, \{B_1, \dots, B_n\}, t$)
(Figure 10.1c).

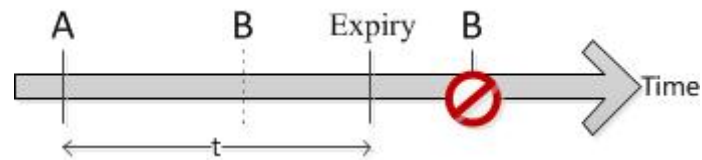
Deadline($A, \{B_1, \dots, B_n\}, t$) means that one of the response events ($B_1..B_n$) must occur within time t of trigger event (A) occurring. Figure 10.1a provides a diagrammatic illustration of the relationship between events A and B and the deadline. In the case of delay (Figure 10.1b), all of the response events can only happen if delay period has been passed following an occurrence of the trigger event. Finally the expiry means that none of the response events can happen after the expiry period has been passed following an occurrence of the trigger event (Figure 10.1c).



(a) Timing diagram of Deadline constraint.



(b) Timing diagram of Delay constraint.



(c) Timing diagram of Expiry constraint. Dash line for occurrence of event B before Expiry indicates the non-deterministic nature of it.

Figure 10.1: Timing diagrams of the introduced timing properties.

We give a semantics to our timing constructs by translating them into Event-B variables, invariants, guards and actions. In particular, these timed-Event-B elements constrain the order between trigger event, responses events and the time-progression event (*Tick_Tock*). For example, the Event-B elements that a deadline constraint adds to a machine, will prevent the *Tick_Tock* event from occurring more than t times in between an occurrence of the trigger event and one of its response events' occurrence. The aim of having these annotations is to hide the complexity of encoding timing constraints in an Event-B model from the modeller. As a result, a timed-Event-B machine from modeller point of view, will be a non-timed-Event-B machine plus a list of its timing constraints, declared by the introduced timing properties. Beside, having the annotations, standardizes the process of specifying discrete timing properties in Event-B models.

In the following sections, some rules are defined to encode each of these timing constructs in an Event-B model. In each case we assume there is already a partial order between the trigger event and the corresponding response events. The assumption is that the response events are only enabled if the trigger event has already happened. This ordering assumption for a sequential control flow, is encoded by using boolean flags. As shown in Figure 10.2a, event A sets the boolean variable A as one of its actions, so when variable A has the value of *TRUE*, indicates that event A has already happened. Also, one of the response events' guards has been dedicated to check the occurrence of event A , by evaluating its flag.

It has not been assumed that the trigger and response events will occur only once. Typically the trigger and response events will be part of an iterative loop and the ordering flags will be reset at the end of each iteration by an event.

Modeling Delay

In this section we explain how delay is encoded in an Event-B model. As mentioned before, in order to have discrete time in Event-B a natural number variable is declared to represent the current time value in the machine and an event is added to model the progress of time.

In order to explain how delay is encoded in Event-B, we will go through the process, for a generic trigger event A and some generic alternative response events $B_1 \dots B_n$. In the following refinement patterns of this chapter, let $G_X(c, v)$ be the set of guards and Act_X be the actions of event X , where c denote the constants and v the variables.

A delay constraint is defined as follows:

$$Delay(A, \{B_1, \dots, B_n\}, t). \quad (10.1)$$

```

TIMING
  Delay( $A, \{B_1, \dots, B_n\}, t$ )
EVENT  $A \hat{=}$ 
  WHERE
     $A = FALSE$ 
     $G_A(c, v)$ 
  THEN
     $A := TRUE$ 
     $Act_A$ 
  END
EVENT  $B_x \hat{=}$ 
  WHERE
     $A = TRUE$ 
     $B_x = FALSE$ 
     $G_{B_x}(c, v)$ 
  THEN
     $B_x := TRUE$ 
     $Act_{B_x}$ 
  END

```

(a) Event A and B_x plus a delay

```

EVENT  $A \hat{=}$ 
  WHERE
     $A = FALSE$ 
     $G_A(c, v)$ 
  THEN
     $A := TRUE$ 
     $tA := time$ 
     $Act_A$ 
  END
EVENT  $B_x \hat{=}$ 
  WHERE
     $A = TRUE$ 
     $B_x = FALSE$ 
     $time \geq tA + t$ 
     $G_{B_x}(c, v)$ 
  THEN
     $B_x := TRUE$ 
     $Act_{B_x}$ 
  END
EVENT  $Tick\_Tock \hat{=}$ 
  THEN
     $time := time + 1$ 
  END

```

(b) Encoded delay for Event A and B_x . The Event-B elements in **red**, are those added to encode the specified delay between Event A and event B_x .

Figure 10.2: Pattern for encoding a delay constraint in Event-B.

There are two steps in order to encode a delay constraint in an Event-B machine. First the occurrence time of the trigger event is recorded in a variable (tA). Then in the events which should be delayed (event B_x , $1 \leq x \leq n$), a guard is needed which prevents the trigger event from being enabled before the stated delay period has been passed from the occurrence of the trigger event. In Figure 10.2 a general pattern of delayed trigger-response and the *Tick_Tock* event, in an Event-B model, has been shown.

Figure 10.2a shows the general pattern for an untimed model plus a delay constraint. Figure 10.2b shows the corresponding Event-B pattern with the delay constraint encoded with an action added to the trigger event to recode the occurrence time of A and a guard added to the response events to prevent early occurrence of a response event. As shown in Figure 10.2b, to encode timing properties, adding the *Tick_Tock* event is required too.

Modelling Expiry

Modelling expiry is similar to the delay. Again the first step is to record the occurrence time of the trigger event, and the next step is to guard the response events according to the recorded time and the specified expiry period. Suppose, we want to force timing property 10.2 to the trigger-response pattern which is shown in Figure 10.3a, how the model should be changed to contain a generic expiry is shown in Figure 10.3b.

$$Expiry(A, \{B_1, \dots, B_n\}, t) \quad (10.2)$$

As shown in Figure 10.3, in order to have an expiry on a trigger-response pattern, an action is needed to record the occurrence time in the trigger event (event A), and a guard on the response events to prevent them from happening if the expiry period has been passed.

```

TIMING
  Expiry(A, {B1, ..., Bn}, t)
EVENT A  $\hat{=}$ 
  WHERE
    A = FALSE
    GA(c, v)
  THEN
    A := TRUE
    ActA
  END
EVENT Bx  $\hat{=}$ 
  WHERE
    A = TRUE
    Bx = FALSE
    GBx(c, v)
  THEN
    Bx := TRUE
    ActBx
  END

```

(a) Event *A* and *B*_{*x*} plus an expiry

```

EVENT A  $\hat{=}$ 
  WHERE
    A = FALSE
    GA(c, v)
  THEN
    A := TRUE
    tA := time
    ActA
  END
EVENT Bx  $\hat{=}$ 
  WHERE
    A = TRUE
    Bx = FALSE
    time ≤ tA + t
    GBx(c, v)
  THEN
    Bx := TRUE
    ActBx
  END
EVENT Tick_Tock  $\hat{=}$ 
  THEN
    time := time + 1
  END

```

(b) Encoded expiry for Event *A* and *B*_{*x*}. The Event-B elements in **red**, are those added to encode the specified expiry between Event *A* and event *B*_{*x*}.

Figure 10.3: Pattern for encoding an expiry constraint in Event-B.

Modelling Deadline

In order to encode expiry and delay, just the trigger and the response events are involved. However, in order to model a deadline the *Tick_Tock* event is involved as well. If the trigger event has happened, we want to force the response event to occur, before passing the deadline. Guarding the *Tick_Tock* event is a way of enforcing one of the $B_1..B_n$ events to occur before passing the deadline.

Suppose, a deadline has been declared as follow:

$$Deadline(A, \{B_1, \dots, B_n\}, t). \quad (10.3)$$

In order to model this constraint in an Event-B model, first the occurrence time of event A should be recorded by adding a new action. Then a guard on the *Tick_Tock* event is needed, to enforce the deadline. In Figure 10.4, how deadline 10.3 can be added to a standard Event-B model is shown. The guard added to the *Tick_Tock* event in Figure 10.4 specifies that if trigger A has occurred but a response B_x has yet to occur then the deadline should not yet reached ($time + 1 \leq tA + t$).

Multiple deadline constraints may be added to a model. In this case, a deadline guard similar to what has been shown in Figure 10.4b should be added to the *Tick_Tock* event for each deadline constraint. Guarding the *Tick_Tock* event can cause a deadlock which will be discuss in the next section in more detail.

10.1.2 Causing Deadlock by Disabling Tick_Tock Forever

It is possible to cause a deadlock by declaring a longer delay between two events than an existing deadline between them or having a deadline on response events that will not be enabled eventually, after the trigger event's occurrence. There are two approaches to detect this kind of deadlock, either by running a model checker (e.g. ProB) and checking the uncovered events or by proving a proof obligation called DDLF (for deadline deadlock freedom) for *Tick_Tock* event. Before explaining the DDLF proof obligation it is useful to take a brief look at deadlock freedom proof obligation by Abrial [?] which inspired DDLF.

Deadlock Freedom Proof Obligation

By guarding the events of a system model, it is possible that the model deadlocks when none of the guards are true. Sometimes, this is a desirable

```

TIMING
  Deadline(A, {B1, ..., Bn}, t)
EVENT A  $\hat{=}$ 
  WHERE
    A = FALSE
    GA(c, v)
  THEN
    A := TRUE
    ActA
  END
EVENT Bx  $\hat{=}$ 
  WHERE
    A = TRUE
    Bx = FALSE
    GBx(c, v)
  THEN
    Bx := TRUE
    ActBx
  END

```

(a) Event *A* and *B*_{*x*} plus a deadline

```

EVENT A  $\hat{=}$ 
  WHERE
    A = FALSE
    GA(c, v)
  THEN
    A := TRUE
    tA := time
    ActA
  END
EVENT Bx  $\hat{=}$ 
  WHERE
    A = TRUE
    Bx = FALSE
    GBx(c, v)
  THEN
    Bx := TRUE
    ActBx
  END
EVENT Tick_Tock  $\hat{=}$ 
  WHERE
    A = TRUE  $\wedge$ 
    (B1 = FALSE  $\wedge$  ..  $\wedge$  Bn = FALSE)  $\Rightarrow$ 
    time + 1  $\leq$  tA + t
  THEN
    time := time + 1
  END

```

(b) Encoded deadline for Event *A* and *B*_{*x*}. The Event-B elements in **red**, are those added to encode the specified deadline between Event *A* and event *B*_{*x*}.

Figure 10.4: Pattern for encoding a deadline constraint in Event-B.

behaviour of the system, but if one of the system specification is the deadlock freeness, it should be verified that there will be an enable event all through the life cycle of the system. In DFL (10.4), for a model with constants c , set of axioms $A(c)$, and set of invariants $I(c, v)$, we prove that one of the guards $G_1(c, v), \dots, G_m(c, v)$ of the various events is always held.

$$\begin{array}{l}
A(c) \\
I(c, v) \\
\vdash \\
G_1(c, v) \vee \dots \vee G_m(c, v)
\end{array} \tag{10.4}$$

By proving DLF for a machine, we prove that there is always an enable event in that machine.

Deadline Deadlock Freedom Proof Obligation (DDL F)

DDL F is not as general as DLF. It aims to check the deadlock property on the *Tick_Tock* event just for the time a deadline period is going to finish. By having DDL F, we want to prove that by the end of a deadline ($time = tA + t$), if *Tick_Tock* event has been disabled by it then at least one of its restricted events is enable. For deadline 10.3 DDL F proof obligation will be as follow:

$$\begin{array}{l}
time = tA + t \\
A = TRUE \wedge B_1 = FALSE \wedge .. \wedge B_n = FALSE \\
\vdash \\
G_{B_1}(c, v) \vee \dots \vee G_{B_n}(c, v)
\end{array} \tag{10.5}$$

By adding DDL F proof obligation 10.5 to timed-Event-B, modellers will be able to check whether they cause a deadlock in a machine by their timing constraints. As a result, if a deadline is causing a deadlock, its DDL F proof obligation cannot be discharged for the *Tick_Tock* event. The most abstract model of a real-time system can be seen as a clock ticking forever. Then by each refinement, new behaviours will be added to it based on the system specifications. By, DDL F we want to prove that, if the *Tick_Tock* event has been disabled by a deadline it will be enabled eventually.

In future work we aim to provide a formal justification for the validity of the DDL F proof obligation in terms of the general DLF obligation.

10.1.3 Some Patterns to Refine Abstract Timing Properties

In this section, some patterns to refine an abstract deadline to more detailed timing constraints will be explained. It should be mentioned that these are not modelling patterns, rather they are refinement patterns; the aim of our patterns is to explain how timing properties can be refined based on some specific control flow refinement patterns. In each of the following sections, the corresponding control flow refinement which makes the timing refinement necessary will be explained and then the timing refinement will be discussed. Besides, invariants required to discharged refinement proof obligations of the timing properties will be discussed for each pattern. Each refinement pattern will be explained by applying it to a generic control flow refinement patterns. In the rest of this report timed-Event-B models will be shown from a modeller point of view. So, each timed-machine will be a list of its timing properties specified by introduced annotations in Section 10.1.1 plus the non-timed Event-B machine. In each refinement pattern, for the constants c and set of variables v of a machine, $G_X(c, v)$ presents the guards of event X in that machine and Act_X presents the actions of that event.

Refining a Deadline to Sequential Sub-Deadlines

Consider an abstract model of a system where there is a deadline between event A and event B . As shown in Figure 10.5, event B can only occur if event A has already happened. The deadline property of this level of abstraction, is shown in Figure 10.6a. In the next refinement event B has been broken to two sequential steps, as shown in Figure 10.5. By breaking event B to B_1 followed by B_2 , its related deadline needs to be broken too. The other important issue in this pattern is that, the abstract event has been refined by the second step, because the accomplishment of the second step is equivalent to accomplishment of the abstract event(B). So the first step should refine *skip*.

Now, in order to respond to the trigger event, two steps have to be accomplished, where each of them has its own deadline. In the concrete level, the trigger event of the deadline constraint for event B_1 is event A and the trigger event for the deadline of event B_2 is event B_1 . Hence, the abstract deadline should be broken to two new deadlines as shown in Figure 10.6b where their combination based on the control flow does not violate the abstract deadline ($t1 + t2 \leq t$).

We need to prove that the concrete machine refines the behaviour of its abstract one. For the refinement pattern, presented in Figure 10.6, it is

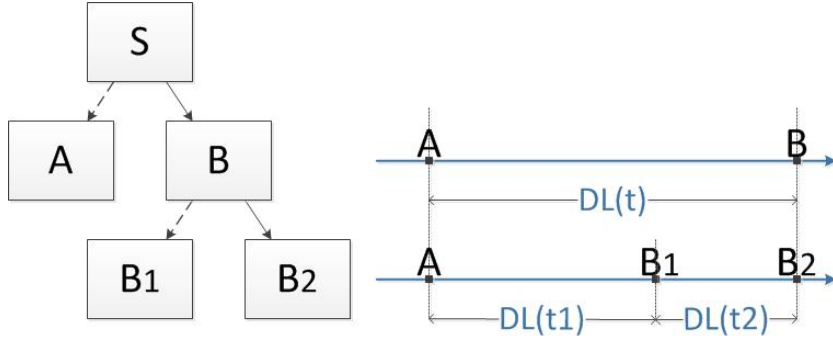


Figure 10.5: Refining an abstract deadline to two sub-deadlines. $DL(x)$ to a deadline constraints with a period of x .

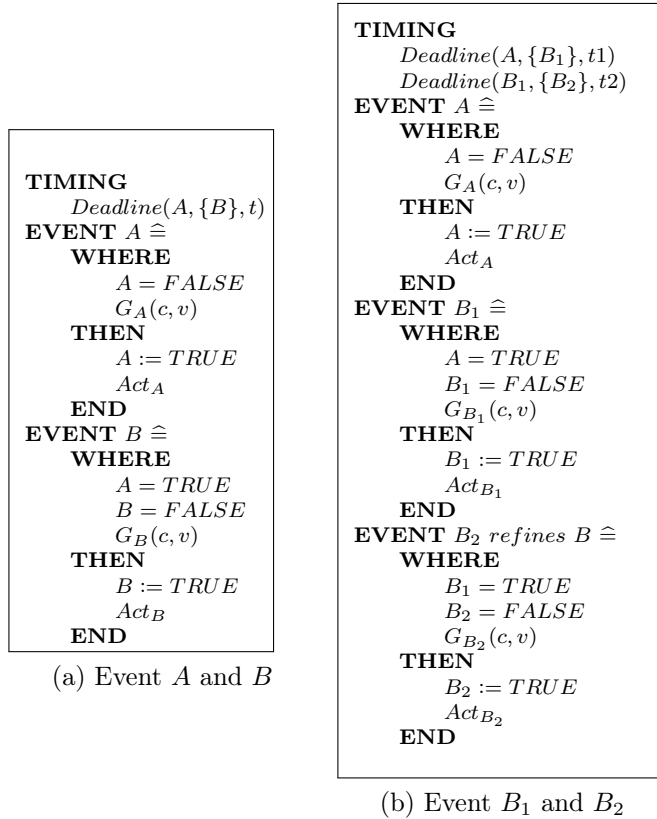


Figure 10.6: Events A and B in plus their deadline property in the abstract Machine in 10.6a followed by events A , events B_1 and B_2 in the concrete machine with their concrete timing properties in 10.6b.

necessary to prove the abstract deadline is held in the concrete machine.

$$Deadline(A, \{B\}, t). \quad (10.6)$$

In the concrete machine, event B has been refined by event B_2 and in the refinement the abstract deadline between A and B is divided into 2 sub-deadlines between A and B_1 and between B_1 and B_2 .

$$Deadline(A, \{B_1\}, t1) \quad (10.7)$$

$$Deadline(B_1, \{B_2\}, t2) \quad (10.8)$$

Based on deadline 10.8 if event B_1 has happened and event B_2 has not happened yet, then the current value of time should be less than or equal to the occurrence time of event B_1 plus the deadline period ($t1$). By this timing constraint the relation of occurrence time of event B_2 and event B_1 has been specified. But we are interested on the relation of the occurrence time of event B_2 and event A . So it is enough to specify the relation of the occurrence time of event B_1 and event A .

The relation between the concrete states and the abstract ones is expressed by *gluing invariants* [?] in Event-B, in order to verify the refinement. Two kinds of gluing invariants are needed, in order to prove that the concrete deadlines satisfy their abstract. The first type is needed to clarify the relation between the order abstract machine and the order in the concrete machine between the involved events of the deadline which has not been modelled by explicit guards. For example based on the guards in this section refinement pattern, B_2 can only happen after B_1 and B_2 can only happen after A , accordingly B_2 can only happen after A but, it has not been mentioned explicitly in its guard. The other type is needed to specify the relation between the new deadlines in the concrete machine and the abstract deadline. In the explained pattern these invariants should be as follow:

- The relation between abstract event and its refining event (B_2 and B are the boolean variables which act as the occurrence flag of events B_2 and B):

$$B_2 = B, \quad (10.9)$$

- The order between concrete events:

$$B_1 = TRUE \Rightarrow A = TRUE, \quad (10.10)$$

- The relation between the abstract deadline trigger time and its concrete one (tA is an integer variable which records the occurrence time of event A and tB_1 does the same thing for event B_1):

$$B_1 = TRUE \Rightarrow tB_1 \leq tA + t1, \quad (10.11)$$

$$A = TRUE \wedge B_1 = FALSE \Rightarrow time \leq tA + t1. \quad (10.12)$$

Invariant (10.9) specifies that the occurrence of event B_2 is equivalent to the occurrence of event B by relating the occurrence flags of event B and event B_2 .

As mentioned, to prove that the concrete deadlines refine the abstract one for the refinement pattern of Figure 10.6, we need to specify the relation of the occurrence time of event B_1 and event A which has been done by the gluing invariant 10.11 based on deadline 10.7.

In event B_1 there is no information about the occurrence time of event A , and invariant 10.11 specifies a relation between the occurrence time of event B_1 and occurrence time of event A . It specifies that if B_1 has occurred, then the time period between A and B_1 does not exceed $t1$. Invariant 10.12 which is equivalent to the required guard on the *Tick_Tock* event for deadline 10.7 provides the required information to discharge the proof obligation of invariant 10.11 for event B_1 .

It should be mentioned that the abstract deadline can be broken into more than two sub-deadlines either by successive refinement steps or by refining the abstract event with more than two sub-sequential events in one refinement step, and for their deadline refinement it is possible to follow a similar approach.

Refining An Abstract Deadline to Alternative Sub-deadlines Case One

It is often the case in system specification, that we require a response to a request within a specific period of time. This response may be successful satisfaction of the request or an alternative way of satisfying the response or an error in the case that the request cannot be met. This kind of behaviour can be modelled by a deadline between the request and the possible responses.

For instance, consider the case where instead of refining event B , in the refinement pattern of Section 10.1.3, by two sequential sub-steps, it has been refined by two alternative events, B_1 and B_2 as shown in Figure 10.7, where event B_1 represents the main response scenario and event B_2 represents the alternative one.

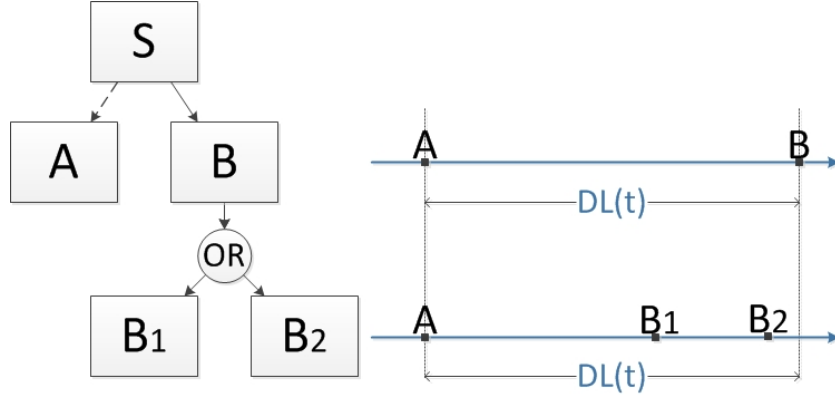


Figure 10.7: Refining an event to two possible events

Based on the refinement, either of event B_1 occurrence or event B_2 occurrence is equivalent to occurrence of abstract event B . So the abstract deadline between event A and event B will be satisfied by occurrence of either of the refining events. As shown in Figure 10.8b, the concrete deadline is between occurrence of event A and either the occurrence of event B_1 or event B_2 , and the period of it is the same of its abstract.

The only kind of invariant required to discharge refinement proof obligations of timing properties in this pattern of refining timing properties, specifies the relation between occurrences of abstract and concrete events. In the this generic refinement pattern this invariant will be as follow:

$$B_2 = TRUE \vee B_1 = TRUE \Leftrightarrow B = TRUE. \quad (10.13)$$

Based on invariant 10.13 either event B_1 or event B_2 occurrence is equivalent to the occurrence of event B .

```

TIMING
  Deadline(A, {B}, t)
EVENT A  $\hat{=}$ 
  WHERE
    A = FALSE
    GA(c, v)
  THEN
    A := TRUE
    ActA
  END
EVENT B  $\hat{=}$ 
  WHERE
    A = TRUE
    B = FALSE
    GB(c, v)
  THEN
    B := TRUE
    ActB
  END

```

(a) Events *A* and *B* and their timing constraint.

```

TIMING
  Deadline(A, {B1, B2}, t)
EVENT A  $\hat{=}$ 
  WHERE
    A = FALSE
    GA(c, v)
  THEN
    A := TRUE
    ActA
  END
EVENT B1 refines B  $\hat{=}$ 
  WHERE
    A = TRUE
    B1 = FALSE
    GB1(c, v)
  THEN
    B1 := TRUE
    ActB1
  END
EVENT B2 refines B  $\hat{=}$ 
  WHERE
    A = TRUE
    B2 = FALSE
    GB2(c, v)
  THEN
    B2 := TRUE
    ActB2
  END

```

(b) Events *A*, *B₁* and *B₂* and their timing constraint

Figure 10.8: Refining a trigger-response pattern and its timing constraints to two alternative responses and required concrete timing properties.

Refining An Abstract Deadline to Alternative Sub-deadlines Case Two

Sometimes in the refinement a request-response sequence in the abstract machine should be refined by several alternative cases of request-response. For example, consider the case where in the abstract model, the car direction is given by the driver and the car will go to that direction. Then it will be refined by two possible cases where either driver sets the direction to left and car goes to left, or driver sets the direction to right and car goes to right.

The difference between this case and the refinement pattern, explained in Section 10.1.3, is just about the trigger event. In Section 10.1.3 the trigger event has not been refined, but in this case the trigger event has been refined to several alternative cases and each concrete trigger event is related to one of the alternative responses. Consider the generic refinement pattern of Section 10.1.3 where event A is refined by two alternative events A_1 and A_2 as shown in Figure 10.9. In this refinement, event A_1 triggers event B_1 and event A_2 triggers event B_2 . As a result, the abstract deadline between event A and event B will be refined as shown in Figure 10.10b.

As shown in Figure 10.10, based on the control flow refinement, the abstract deadline will be replaced with new deadlines for each alternative case of request-response. Beside, you can see the enforcement of the partial order between each response and its corresponding request by using boolean flags in Figure 10.10b.

The only difference between concrete deadlines and the abstract one, is the name of trigger and response events. For each case of request-response the corresponding request event and response event will be constraint by a deadline with a same period as the abstract one. To discharge the refinement proof obligations of timing properties, a gluing invariant is required for each concrete sequence of request-response, to specify the relation between occur-

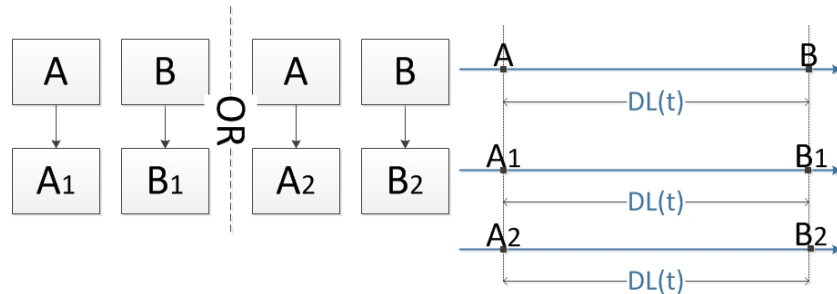


Figure 10.9: How a single request-response sequence can be refined to several request-response cases.

```

TIMING
  Deadline(A, {B}, t)
EVENT A  $\hat{=}$ 
  WHERE
    A = FALSE
    GA(c, v)
  THEN
    A := TRUE
    ActA
  END
EVENT B  $\hat{=}$ 
  WHERE
    A = TRUE
    B = FALSE
    GB(c, v)
  THEN
    B := TRUE
    ActB
  END

```

(a) Events *A* and *B* and their timing constraint.

```

TIMING
  Deadline(A1, {B1}, t)
  Deadline(A2, {B2}, t)
EVENT A1 refines A  $\hat{=}$ 
  WHERE
    A1 = FALSE
    GA1(c, v)
  THEN
    A1 := TRUE
    ActA1
  END
EVENT B1 refines B  $\hat{=}$ 
  WHERE
    A1 = TRUE
    B1 = FALSE
    GB1(c, v)
  THEN
    B1 := TRUE
    ActB1
  END
EVENT A2 refines A  $\hat{=}$ 
  WHERE
    A2 = FALSE
    GA2(c, v)
  THEN
    A2 := TRUE
    ActA2
  END
EVENT B2 refines B  $\hat{=}$ 
  WHERE
    A2 = TRUE
    B2 = FALSE
    GB2(c, v)
  THEN
    B2 := TRUE
    ActB2
  END

```

(b) Events *A1*, *A2*, *B1* and *B2* and their timing constraint

Figure 10.10: Refining a trigger-response pattern and its timing constraints to two alternative request-response cases and required concrete timing properties of them.

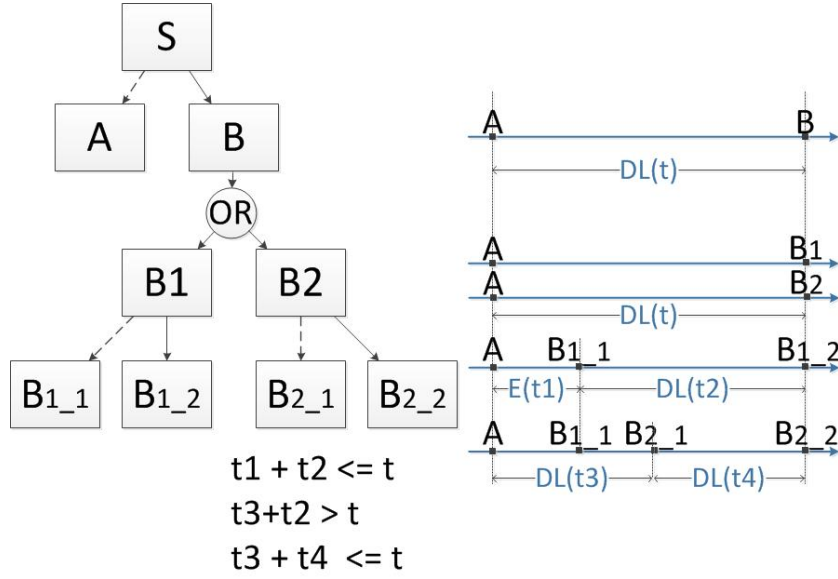


Figure 10.11: Refining each of the alternative scenarios to two sequential steps

rences of the abstract events and their concrete ones. Based on our generic refinement pattern the required invariants should be as follow:

$$A1 = TRUE \vee A2 = TRUE \Leftrightarrow A = TRUE \quad (10.14)$$

$$B_1 = TRUE \vee B_2 = TRUE \Leftrightarrow B = TRUE \quad (10.15)$$

So in this way it is possible to break an abstract deadline for a request-response sequence to several alternative deadlines when it has been refined by several alternative request-response sequences.

Refining Alternative Sub-Deadlines by Sequential Sub-Deadlines and Expiries

In order to explain this pattern, the refinement pattern of Section 10.1.3 will be continued. So, in the current state, we have a trigger event A and two alternative responses, event B_1 and B_2 . The timing properties of these levels of abstraction, have been shown in Figure 10.8.

In the next refinement, each of event B_1 and event B_2 will be refined to two sequential steps and their deadline will be refined to two sequential deadlines, as in the pattern shown in Section 10.1.3 (event B_1 will be broken to events B_{1_1} and B_{1_2} and event B_2 will be broken to events B_{2_1} and B_{2_2}).

TIMING	
<i>Expiry</i> ($A, B_{1\_1}, t1$)	
<i>Deadline</i> ($B_{1\_1}, \{B_{1\_2}\}, t2$)	
<i>Deadline</i> ($A, \{B_{1\_1}, B_{2\_1}\}, t3$)	
<i>Deadline</i> ($B_{2\_1}, \{B_{2\_2}\}, t4$)	
EVENT $B_{1\_1} \hat{=}$	EVENT $B_{2\_1} \hat{=}$
WHERE	WHERE
$A = TRUE$	$A = TRUE$
$B_{1\_1} = FALSE$	$B_{2\_1} = FALSE$
$G_{B_{1\_1}}(c, v)$	$G_{B_{2\_1}}(c, v)$
THEN	THEN
$B_{1\_1} := TRUE$	$B_{2\_1} := TRUE$
$Act_{B_{1\_1}}$	$Act_{B_{2\_1}}$
END	END
EVENT B_{1_2} refines $B_1 \hat{=}$	EVENT B_{2_2} refines $B_2 \hat{=}$
WHERE	WHERE
$A = TRUE$	$A = TRUE$
$B_{1\_1} = TRUE$	$B_{2\_1} = TRUE$
$B_{1\_2} = FALSE$	$B_{2\_2} = FALSE$
$G_{B_{1\_2}}(c, v)$	$G_{B_{2\_2}}(c, v)$
THEN	THEN
$B_{1\_2} := TRUE$	$B_{2\_2} := TRUE$
$Act_{B_{1\_2}}$	$Act_{B_{2\_2}}$
END	END

Figure 10.12: Events B_{1_1} , B_{1_2} , B_{2_1} and B_{2_2} of the final refinement and their timing properties.

In this system, initially either of response cases (represented by event B_1 and event B_2) are enable after the occurrence of the request event, but if the first step (modelled by event B_{1_1}) of the response case modelled by event B_1 has not been accomplished within $t1$ time after occurrence of event A , the response case modelled by event B_2 will be the only possible response and its first step (modelled by event B_{2_1}) has to happen before the specified deadline ($t3$). As a result by the first deadline in the concrete machine, either the first response case has been activated or the second one (by occurrence of their first steps). For the next step, occurrence of event B_{1_1} enables event B_{1_2} , and occurrence of event B_{2_1} enables event B_{2_2} as shown in Figures 10.11.

It is assumed that the sequence of the concrete deadline between event A and event B_{1_1} (*Deadline*($A, \{B_{1\_1}, B_{2\_1}\}, t3$)), followed by the concrete deadline between event B_{1_1} and B_{1_2} (*Deadline*($B_{1\_1}, \{B_{1\_2}\}, t2$)), does not refine the abstract deadline between event A and event B ($t3 + t2 > t$). This situation has been caused by having sequential sub-deadlines on alternative control flow sequences, where each of those control flow sequences has other timing constraints on its own. In this refinement pattern, there are two alternative sequences, event B_{1_1} followed by event B_{2_1} and event B_{1_2} followed by event B_{2_2} .

As mentioned above, event B_{1_1} has an expiry constraint too. So after a

specific time, it cannot happen any more and the only possible response will be the alternative response modelled by events B_{2_1} and B_{2_2} .

By enforcing this constraint with an expiry as shown in Figure 10.12, the concrete timing properties will satisfy their abstract ones. By having an expiry constraint between events A and B_{1_1} for the period of $t1$, it will be guaranteed that if event B_{1_1} has occurred, $t1$ time units have been passed from event A occurrence (Formulated in invariant 10.16).

$$B_{1\_1} = TRUE \Rightarrow B_{1\_1}t \leq tA + t1 \quad (10.16)$$

From event B_{1_1} occurrence, event B_{1_2} has $t2$ time units to happen based on the existing deadline between them. As a result, the abstract deadline has been contained by the concrete timing properties. This pattern shows how combination of deadline and expiry can be useful in modelling and refining the timing properties of a time-critical system.

Using Alternative Deadlines Instead of Combining Deadline and Expiry

Some may ask why we do not use two disjunctive deadlines instead of a deadline and an expiry for timing properties similar to events A , B_{1_1} and B_{1_2} in the example of Section 10.1.3. To explain it, we will go through the refinement process of an abstract deadline by two disjunctive deadlines on a same control flow refinement pattern. Suppose we want to encode timing properties of events A , B_{1_1} and B_{2_1} as follow:

$$Deadline(A, \{B_{1\_1}\}, t1) \vee Deadline(A, \{B_{2\_1}\}, t3) \quad \textbf{Where } t1 < t3 \quad (10.17)$$

To encode these properties in an Event-B model, the required guard on *Tick_Tock* event will be as follow based on our approach explained in Section 10.1.1:

$$(A = TRUE \wedge B_{1\_1} = FALSE \Rightarrow time + 1 \leq tA + t1) \quad \vee \\ (A = TRUE \wedge B_{2\_1} = FALSE \Rightarrow time + 1 \leq tA + t3) \quad (10.18)$$

In the following we will prove that deadline guard 10.18 is equivalent to the required deadline guard for deadline 10.19 and as a result, it does not enforce the expiry property to the model.

$$Deadline(A, \{B_{1\_1}, B_{2\_1}\}, t3) \quad (10.19)$$

Based on the deadline encoding approach explained in Section 10.1.1 the required guard on the *Tick_Tock* event for deadline 10.19 is as follow:

$$A = TRUE \wedge B_{1\_1} = FALSE \wedge B_{2\_1} = FALSE \Rightarrow time + 1 \leq tA + t3 \quad (10.20)$$

First we need to prove one of the required hypothesis as follow:

$$\begin{aligned}
& t3 > t1 \\
& time \geq 0 \\
& tA \geq 0 \\
& \vdash \\
& time + 1 \leq tA + t1 \Rightarrow time + 1 \leq tA + t3 \quad (10.21)
\end{aligned}$$

Next step is to simplify Deadline guard 10.18 as follow:

$$\begin{aligned}
& (A = TRUE \wedge B_{1\_1} = FALSE \Rightarrow time + 1 \leq tA + t1) \vee \\
& (A = TRUE \wedge B_{2\_1} = FALSE \Rightarrow time + 1 \leq tA + t3) \\
& = \quad \text{Implicationdefinition} \\
& \neg (A = TRUE \wedge B_{1\_1} = FALSE) \vee time + 1 \leq tA + t1 \vee \\
& \neg (A = TRUE \wedge B_{2\_1} = FALSE) \vee time + 1 \leq tA + t3 \\
& = \quad \text{DeMorgan} \\
& A \neq TRUE \vee B_{1\_1} \neq FALSE \vee time + 1 \leq tA + t1 \vee \\
& A \neq TRUE \vee B_{2\_1} \neq FALSE \vee time + 1 \leq tA + t3 \\
& = \quad \text{Absorption} \\
& A \neq TRUE \vee B_{1\_1} \neq FALSE \vee time + 1 \leq tA + t1 \vee \\
& B_{2\_1} \neq FALSE \vee time + 1 \leq tA + t3 \quad (10.22)
\end{aligned}$$

Also, predicate 10.21 can be simplified as follow:

$$\begin{aligned}
& time + 1 \leq tA + t1 \Rightarrow time + 1 \leq tA + t3 \\
& = \quad \text{Implicationdefinition} \\
& \neg (time + 1 \leq tA + t1) \vee time + 1 \leq tA + t3 \quad (10.23)
\end{aligned}$$

By assuming predicates 10.22 and 10.23 and applying resolution rule the following can be concluded:

$$\begin{aligned}
& A \neq TRUE \vee B_{1\_1} \neq FALSE \vee time + 1 \leq tA + t1 \vee \\
& B_{2\_1} \neq FALSE \vee time + 1 \leq tA + t3 \\
& \neg (time + 1 \leq tA + t1) \vee time + 1 \leq tA + t3 \\
& \vdash \\
& A \neq TRUE \vee B_{1\_1} \neq FALSE \vee B_{2\_1} \neq FALSE \vee time + 1 \leq tA + t3 \quad (10.24)
\end{aligned}$$

By applying the Implication Definition and De Morgan rules to predicate 10.24 the following predicate will be gained:

$$\begin{aligned}
& A \neq TRUE \vee B_{1\_1} \neq FALSE \vee B_{2\_1} \neq FALSE \vee time + 1 \leq tA + t3 \\
& = \quad \text{Implicationdefinition} \\
& \neg (A \neq TRUE \vee B_{1\_1} \neq FALSE \vee B_{2\_1} \neq FALSE) \Rightarrow time + 1 \leq tA + t3 \\
& = \quad \text{DeMorgan} \\
& A = TRUE \wedge B_{1\_1} = FALSE \wedge B_{2\_1} = FALSE \Rightarrow time + 1 \leq tA + t3
\end{aligned}$$

As proved above, deadline guard 10.18 is equivalent to deadline guard 10.20. Accordingly, alternative deadlines 10.17 allow event B_{1_1} to occur after $t1$ time units from occurrence of event A . As a result by having two disjunctive deadlines, the expiry constraint on event B_{1_1} will not be enforced.

In this section some approaches have been introduced in order to refine timing properties of a timed-Event-B model. These patterns do not contain all the possible cases of refining timing properties and we are still working on other possible refinement patterns.

10.1.4 Encoding a Sequential Control Flow in Event-B

As explained in Section 10.1.1 our approach to model timing properties in Event-B is based on the assumption that sequential control flow, is encoded by using boolean flags. This assumption is the result of our investigation on existing approaches to model a sequential order in an Event-B model.

The sequential order between trigger events and their response events can be modelled by a set too. It is possible to dedicate a constant to each event and declare a set which is empty in the initialization and occurrence of each event causes its corresponding constant to be added to the mentioned set as shown in Figure 10.13b for an order between event A and event B . Based on this approach, if event B has to happen after event A , then event B has to check if the corresponding constant of event A has been already added to the set.

The problem of modelling sequential order in this way is refinement. If a *skip* event is added in a refinement, a new set has to be declared, because it is not possible to manipulate an abstract variable in a *skip* event. As shown in Figure 10.14b because of adding a new event (*prefB*) between events A and B in the refinement, the previous set of occurred events (*order*) has been replaced by a new one (*newOrder*).

Declaring a new ordering set requires the modeller to prove the relation between the abstract ordering set and the concrete one for all the possible members of these two sets, which based on extent of the refinement and

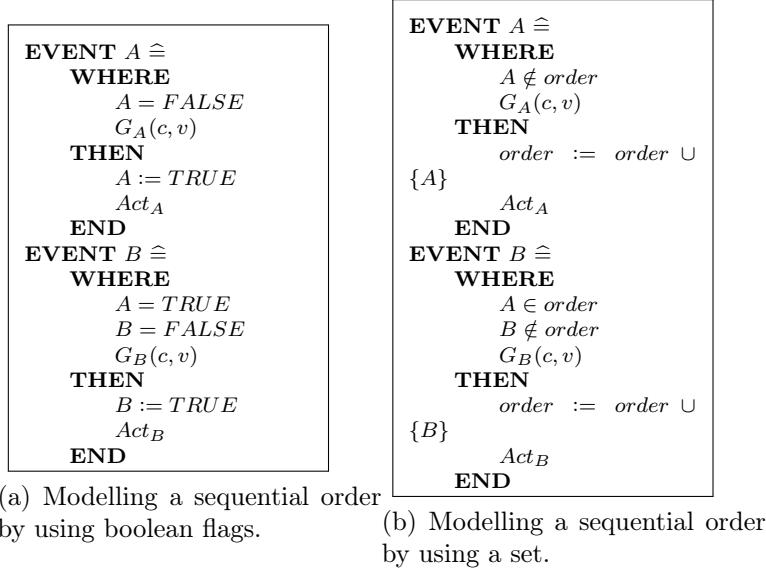


Figure 10.13: Two existing approaches to model a sequential order has been shown for an order between generic events A and B

number of existing events in the machine, can be a time consuming process. For the control flow refinement pattern where the order between events A and B has been refined to an order between events A , $preB$ and B , the following invariants are required besides the type invariant for the new ordering set ($newOrder$), to discharge the refinement proof obligations:

$$newOrder \setminus \{preB\} \subseteq order \quad (10.26)$$

$$preB \in newOrder \Rightarrow A \in newOrder \quad (10.27)$$

Invariant 10.26 specifies the relation between occurrence of the refining events and their abstracts, and invariant 10.27 specifies the relation between occurrence of new event ($preB$) and refining events. Besides, modeller needs to replace the abstract ordering set with the new one in the guards and the actions of all the refining events.

But in the boolean flag approach, each flag is independent from the other, and adding a new one or refining an existing one will not effects others. By adding a new event, no changes is required on other events unless their order has been changed and the related guard should be replaced with a new one based on the new order. For Figure 10.14a refinement pattern the only required invariant to discharge the refinement proof obligations is:

$$preB = TRUE \Rightarrow A = TRUE \quad (10.28)$$

<pre> EVENT <i>A refines A</i> $\hat{=}$ WHERE <i>A</i> = <i>FALSE</i> $G_A(c, v)$ THEN <i>A</i> := <i>TRUE</i> <i>Act_A</i> END EVENT <i>preB</i> $\hat{=}$ WHERE <i>A</i> = <i>TRUE</i> <i>preB</i> = <i>FALSE</i> $G_{preB}(c, v)$ THEN <i>preB</i> := <i>TRUE</i> <i>Act_{preB}</i> END EVENT <i>B refines B</i> $\hat{=}$ WHERE <i>preB</i> = <i>TRUE</i> <i>B</i> = <i>FALSE</i> $G_B(c, v)$ THEN <i>B</i> := <i>TRUE</i> <i>Act_B</i> END </pre>	<pre> EVENT <i>A refines A</i> $\hat{=}$ WHERE <i>A</i> $\notin$ <i>newOrder</i> $G_A(c, v)$ THEN <i>newOrder</i> := <i>newOrder</i> $\cup$ {<i>A</i>} <i>Act_A</i> END EVENT <i>preB</i> $\hat{=}$ WHERE <i>A</i> $\in$ <i>newOrder</i> <i>preB</i> $\notin$ <i>newOrder</i> $G_{preB}(c, v)$ THEN <i>newOrder</i> := <i>newOrder</i> $\cup$ {<i>preB</i>} <i>Act_{preB}</i> END EVENT <i>B refines B</i> $\hat{=}$ WHERE <i>preB</i> $\in$ <i>newOrder</i> <i>B</i> $\notin$ <i>newOrder</i> $G_B(c, v)$ THEN <i>newOrder</i> := <i>newOrder</i> $\cup$ {<i>B</i>} <i>Act_B</i> END </pre>

(a) Refining a sequential order modelled by boolean flags. (b) Refining a sequential order modelled by a set.

Figure 10.14: Effect of adding a *skip* event in a refinement to a sequential order, modelled by using a occurrence history set.

As shown in these two examples, by modelling sequential orders by set instead of boolean flags, besides the invariants for specifying the relation between the occurrence of the new events and the occurrence of the refining events, we also need some invariants to show the relation between the occurrence of the refining events and their abstract ones which will increase the complexity of the refinement prove.

Requiring less changes and gluing invariants for the control flow refinement and causing less dependency between events by using boolean variables to model a sequential order between events in an Event-B model have convinced us to assume this approach to encoding timing properties in Event-B.

10.1.5 Achievements

In this chapter three groups of timing properties have been defined and some annotations have been introduced to express them based on events' occurrences. Then how we encoded them in Event-B were explained and some patterns to refine those timing properties based on some specific control flow refinements have been explained. In the following Chapter how these features are used to add timing specification to an automatic gear controller case study will be explained briefly.

10.2 Towards Probabilistic Modelling in Event-B

System development by refinement is a formalised model-driven approach to developing complex systems. Refinement enables correct-by-construction development of systems. Its top-down development paradigm allows us to cope with system complexity via abstraction, gradual model transformation and proofs. Currently the use of refinement is mainly limited to reasoning about functional correctness. Meanwhile, in the area of dependable system development – the area where the formal modelling is mostly demanded – besides functional correctness it is equally important to demonstrate that the system adheres to certain quantitatively expressed dependability level. Hence, there is a clear need for enhancing formal modelling with a capability of stochastic reasoning about dependability.

In this work [TTL10], we propose an approach to introducing probabilities into Event-B modelling. Our aim is to enable quantitative assessment of dependability attributes, in particular, reliability of systems modelled in Event-B. We consider cyclic systems and show that their behaviour can be represented via a common Event-B modelling pattern. We show then how

to augment such models with probabilities (using a proposed probabilistic choice operator) that in turn would allow us to assess their reliability.

Reliability is a probability of system to function correctly over a given period of time under a given set of operating conditions [?, Vil91, O’C95]. It is often assessed using the classical Markov modelling techniques [KS60]. We demonstrate that Event-B models augmented with probabilities can be given the semantic of a Markov process (or, in special cases, a Markov chain). Then refinement of augmented Event-B models essentially becomes reliability-parametrised development, i.e., the development that not only guarantees functional correctness but also ensures that reliability of refined model is preserved or improved. The proposed approach allows us to smoothly integrate quantitative dependability assessment into the formal system development.

10.2.1 Modelling of Cyclic Systems in Event-B

In this work, we focus on modelling systems with cyclic behaviour, i.e. the systems that iteratively execute a predefined sequence of steps. Typical representatives of such cyclic systems are control and monitoring systems. An iteration of a control system includes reading the sensors that monitor the controlled physical processes, processing the obtained sensor values and setting actuators according to a predefined control algorithm. In principle, the system could operate in this way indefinitely long. However, different failures may affect the normal system functioning and lead to a shutdown. Hence, during each iteration the system status should be re-evaluated to decide whether it can continue its operation.

In general, *operational* states of a system, i.e., the states where system functions properly, are defined by some predicate $J(v)$ over the system variables. Usually, essential properties of the system (such as safety, fault tolerance, liveness properties) can be guaranteed only while system stays in the operational states. The predicate $J(v)$ partitions the system state space S into two disjoint classes of states – *operational* (S_{op}) and *non-operational* (S_{nop}) states, where $S_{op} \hat{=} \{s \in S \mid J.s\}$ and $S_{nop} \hat{=} S \setminus S_{op}$. Please note, that here we consider only the set of *observable* system states, i.e. states where an iteration of a cyclic system can start and terminate.

Abstractly, we can specify a cyclic system in Event-B as shown in Figure 10.15. In the machine CS , the variable st abstractly models the system state, which can be either operational ($J(st)$ is true) or failed ($J(st)$ is false). The event $iter$ abstractly models one iteration of the system execution. As a result of this event, the system can stay operational or fail. In the first case, the system can execute its next iteration. In the latter case, the system

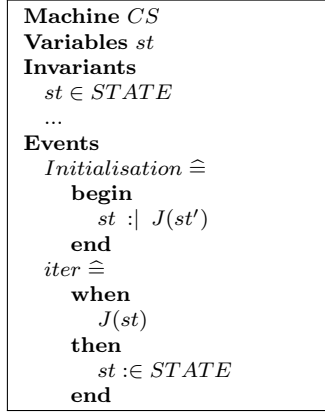


Figure 10.15: Abstract cyclic system

deadlocks.

The **Invariants** clause (besides defining the variable types) can contain other essential properties of the system. Usually they are stated only over the operational states, i.e., they are of the form:

$$J(st) \quad \Rightarrow \quad \dots$$

We can refine the abstract specification CS by introducing specific implementation details. For example, we may explicitly introduce new events modelling the environment as well as reading the sensors or setting the actuators. The event $iter$ can be also refined, e.g., into *detection* operation, which decides whether the system can continue its normal operation or has to shut down due to some unrecoverable failure. However, the Event-B refinement process will preserve the cyclic nature of the system described in the abstract specification CS .

The only other constraint we put on the refinement process is that all the new events introduced in refined models can be only enabled in operational system states, e.g., the event guards should contain the condition $J(v)$. To enforce this constraint, we propose a simple syntactic extension of the Event-B model structure. Specifically, we introduce a new clause **Operational guards** containing state predicates precisely defining the subset of operational system states. This is a shorthand notation implicitly adding the corresponding guard conditions to all events enabled in the operational states (except initialisation). We also assume that, like model invariants, operational guards are inherited in all refined models. By using this new clause, we can rewrite the system CS as follows.

Machine CS
Variables st
Invariants
$st \in STATE$
...
Operational guards
$J(st)$
Events
$Initialisation \hat{=}$
begin
$st := J(st')$
end
$iter \hat{=}$
begin
$st := STATE$
end

Figure 10.16: Cyclic system with an operational guard

In general, the behaviour of some cyclic system M can be intuitively described by the sequential composition ($Initialisation; \mathbf{do} J \rightarrow E \mathbf{od}$), where $\mathbf{do} J \rightarrow E \mathbf{od}$ is a while-loop with the operational guard J and the body E that consists of all the machine events except initialisation. For example, the behaviour of CS can be described simply as ($Initialisation; \mathbf{do} J \rightarrow iter \mathbf{od}$).

Each iteration of the loop maps the current operational system state into a subset of S . The resulting set of states represents all possible states that can be reached due to system nondeterministic behaviour. Therefore, an iteration of a cyclic system M can be defined as a partial function $\mathcal{I}_M$ of the type $S_{op} \rightarrow \mathcal{P}(S)$.¹ The concrete definition of $\mathcal{I}_M$ can be derived from the composition of before-after predicates of the involved events. Moreover, we can also consider the behaviour of the overall system and observe that the final state of every iteration defines the initial state of the next iteration provided the system has not failed.

The specification pattern for modelling cyclic systems defined above restricts the shape of Event-B models. This restriction allow us to propose a scalable approach to integrating probabilistic analysis of dependability into Event-B. This approach we present next.

10.2.2 Stochastic Modelling in Event-B

Hallerstede and Hoang [HH07] have extended the Event-B framework with a new operator – *qualitative probabilistic choice*, denoted $\oplus$. This operator assigns new values to variables with some positive but generally unknown probability. The extension aimed at introducing into Event-B the concept

¹Equivalently, we can define an iteration as a relation between S_{op} and S .

of “almost-certain convergence”—probabilistically certain termination of new event operations introduced by model refinement. The new operator can replace a nondeterministic choice (assignment) statement in the event actions. It has been shown that any probabilistic choice statement always refines its demonic nondeterministic counterpart [MM05]. Hence such an extension is not interfering with traditional refinement process.

We aim at introducing *quantitative* probabilistic choice, i.e., the operator $\oplus$ with precise probabilistic information about how likely a particular choice should be made. In other words, it behaves according to some known probabilistic distribution. The quantitative probabilistic assignment

$$x \oplus | x_1 @ p_1; \dots; x_n @ p_n,$$

where $\sum_{i=1}^n p_i = 1$, assigns to the variable x a new value x_i with the corresponding non-zero probability p_i . Similarly to Hallersted and Hoang, we can introduce probabilistic choice only to replace the existing demonic one.

To illustrate the proposed extension, in Figure 10.17 we present a small example of a probabilistic communication protocol implementing transmission over unreliable channel. Since the channel is unreliable, sent messages may be lost. In the model AM shown on the left-hand side, the occurrence of faults is modelled nondeterministically. Specifically, the variable msg_a is nondeterministically assigned *delivered* or *lost*. In the model AM' , the nondeterministic choice is replaced by the probabilistic one, where the non-zero constant probabilities p and $1 - p$ express how likely a message is getting delivered or lost. According to the theory of probabilistic refinement [MM05], the machine AM' is a refinement of the machine AM . The model refinement relation is denoted $\sqsubseteq$.

Next we show how to define refinement between probabilistic systems modelled in (extended) Event-B. In particular, our notion of model refinement can be specialized to quantitatively demonstrate that the refined system is at least as reliable as its more abstract counterpart.

10.2.3 Reliability Refinement

Fully Probabilistic Systems

Let us first consider fully probabilistic systems, i.e., systems containing only probabilistic branching. The quantitative information present in a probabilistic Event-B model requires lifting the notion of the system state to a probabilistic distribution over it:

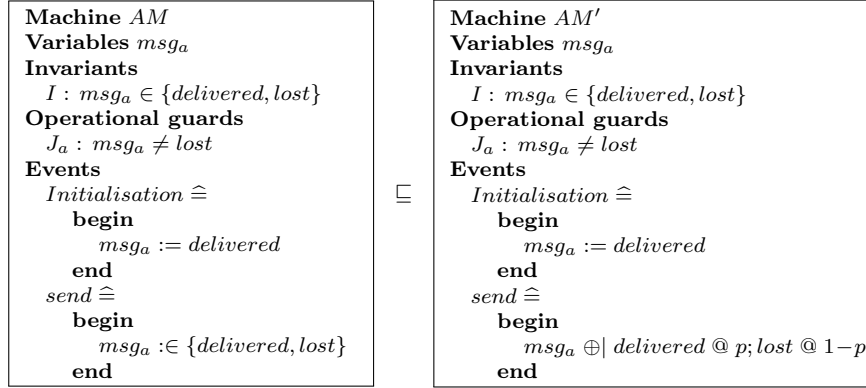


Figure 10.17: A simple communication protocol: introducing probabilities

Definition 1 [Probabilistic distribution]. For a system state space S , the set of distributions over S is

$$\bar{S} \hat{=} \{ \Delta : S \rightarrow [0, 1] \mid \sum_{s \in S} \Delta.s = 1 \},$$

where $\Delta.s$ is the probability of reaching the state s .

Each iteration of a fully probabilistic system then maps some initial operational state to a subset of S according to some probabilistic distribution, i.e., we can define a single iteration $\mathcal{PL}_M$ of a probabilistic cyclic system M as a partial function of the type $S_{op} \rightarrow \bar{S}$.

As it was mentioned before, all elements of system state are partitioned into two disjoint classes of operational and non-operational states. For any state $s \in S_{op}$, its distribution Δ is defined by probabilistic choice statements (assignments) presented in an Event-B machine. However, once the system fails, it stays in the failed (non-operational) state. This means that, for any state $s \in S_{nop}$, its distribution Δ is such that $\Delta.s = 1$ and $\Delta.s' = 0$, if $s' \neq s$.

Once we know the probabilistic state distribution Δ , we can quantitatively assess the probability that the operational guard J is preserved by a single iteration. However, our goal is to evaluate system reliability. In engineering, reliability [Vil91, O'C95] is generally measured by the probability that an entity $\mathcal{E}$ can perform a required function under given conditions for the time interval $[0, t]$:

$$R(t) = \mathbf{P}\{\mathcal{E} \text{ not failed over time } [0, t]\}.$$

Hence reliability can be expressed as the probability that J remains *true* during a certain number of iterations, i.e., the probability of system staying

operational for t iterations:

$$R(t) = \mathbf{P}\{\Box^{\leq t} J\}.$$

Here we use the modal operator $\Box$ borrowed from temporal logic (LTL or (P)CTL, for instance). The formula $(\Box^{\leq t} J)$ means that J holds *globally* for the first t iterations. It is straightforward to see that this property corresponds to the standard definition of reliability given above.

Let M and M' be probabilistic Event-B models of cyclic systems. We strengthen the notion of Event-B refinement by additionally requiring that the refined model will execute more iterations before shutdown with a higher probability:

Definition 2 [Refinement for probabilistic cyclic systems]. *For two (fully) probabilistic Event-B models M and M' of cyclic systems such that $M \triangleq (\text{Initialisation}; \mathbf{do} J \rightarrow E \mathbf{od})$ and $M' \triangleq (\text{Initialisation}'; \mathbf{do} J' \rightarrow E' \mathbf{od})$, we say that M' is a refinement of M , if and only if*

1. M' is an Event-B refinement of M ($M \sqsubseteq M'$), and

2. $\forall t \in \mathbb{N}_1 \cdot \mathbf{P}\{\Box^{\leq t} J\} \leq \mathbf{P}\{\Box^{\leq t} J'\}$.

Remark 1. If the second condition of Definition 2 holds not for all t , but for some interval $t \in 1..T$, $T \in \mathbb{N}_1$, we say that M' is a *partial* refinement of M for $t \leq T$.

Generally, all the non-operational states S_{nop} can be treated as a singleton set, since we do not usually care at which particular state the operational guard has been violated. Therefore, by assuming that $S = \{s_1, \dots, s_n\}$ and $S_{nop} = \{s_n\}$.

Now let us consider the behaviour of some cyclic system M in detail. We can assume that the initial system state (say s_1) belongs to the final set of states $\{s_1, \dots, s_n\}$. This is a state where the system works “perfectly”. After its first iteration, the system goes to some state s_i with the probability $\Delta_1.s_i$ and s_i becomes the current system state. At this point, if $i = n$, system shutdown is initiated. Otherwise, the system starts a new iteration and, as a result, goes to some state s_j with the probability $\Delta_i.s_j$ and so on. It is easy to see that this process is completely defined by the following state transition matrix

$$P_M = \begin{pmatrix} \Delta_{1.s_1} & \Delta_{1.s_2} & \dots & \Delta_{1.s_n} \\ \Delta_{2.s_1} & \Delta_{2.s_2} & \dots & \Delta_{2.s_n} \\ \vdots & \vdots & \ddots & \vdots \\ \Delta_{n.s_1} & \Delta_{n.s_2} & \dots & \Delta_{n.s_n} \end{pmatrix}$$

which in turn unambiguously defines the underlying Markov process (absorbing discrete time Markov chain, to be precise).

Let us note that the state transition matrix of a Markov chain and its initial state allow us to calculate the probability that the defined Markov process (after k steps) will be in a state s_i (see [KS60] for example). Then we can rewrite the second condition of Definition 2 in the following way:

Proposition 1. *For two probabilistic Event-B models M and M' such that $M \hat{=} (\text{Initialisation}; \mathbf{do} J \rightarrow E \mathbf{od})$ and $M' \hat{=} (\text{Initialisation}'; \mathbf{do} J' \rightarrow E' \mathbf{od})$, the inequality*

$$\forall t \in \mathbb{N}_1 \cdot \mathbf{P}\{\Box^{\leq t} J\} \leq \mathbf{P}\{\Box^{\leq t} J'\}$$

is equivalent to

$$\forall t \in \mathbb{N}_1 \cdot ((P_{M'})^t)_{1n'} \leq ((P_M)^t)_{1n}, \quad (10.29)$$

where $S = \{s_1, \dots, s_n\}$ and $S' = \{s_1, \dots, s_{n'}\}$ are the system state spaces of M and M' accordingly, and $(\cdot)_{1n}$ is the (n) -th element of the first row of a matrix.

Proof Directly follows from fundamental theorems of the theory of Markov chains [KS60]. $\blacksquare$

In general, the initial system state is not necessarily the given state s_1 but can be defined by some initial state distribution Δ_0 . In this case the inequality (10.29) should be replaced with

$$([\Delta'_0] \cdot P_{M'}^t)(n') \leq ([\Delta_0] \cdot P_M^t)(n),$$

where $[\Delta_0] = (\Delta_{0.s_1}, \dots, \Delta_{0.s_n})$, $[\Delta'_0] = (\Delta'_{0.s_1}, \dots, \Delta'_{0.s_{n'}})$ and $([\Delta_0] \cdot P_M^t)(n)$ is the n -th component of the column vector $([\Delta_0] \cdot P_M^t)$.

To illustrate our approach to refining fully probabilistic systems, let us revisit our transmission protocol example. To increase reliability of transmission, we refine the protocol to allow the sender to repeat message sending in case of delivery failure. The maximal number of such attempts is given as the predefined positive constant N . The resulting Event-B model CM

```

Machine CM
Variables msgc, att
Invariants
  I1 : msgc ∈ {delivered, try, lost}
  I2 : att ∈ 1..N
Operational guards
  Jc : msgc ≠ lost
Events
  Initialisation ≡
    begin
      msgc := delivered
      att := 1
    end
  start ≡
    when
      msgc = delivered
    then
      msgc := try
    end
  send1 ≡
    when
      msgc = try ∧ att < N
    then
      msgc, att ⊕ (delivered, 1) @ p; (try, att+1) @ 1-p
    end
  send2 ≡
    when
      msgc = try ∧ att = N
    then
      msgc, att ⊕ (delivered, 1) @ p; (lost, att) @ 1-p
    end

```

Figure 10.18: A simple communication protocol: probabilistic refinement

is presented in Figure 10.18. Here the variable *att* represents the current sending attempt. Moreover, the event *send* is split to model the situations when the threshold *N* has been accordingly reached and not reached.

The Event-B machine *CM* can be proved to be a probabilistic refinement of its abstract probabilistic model (the machine *AM'* in Figure 10.17) according to Definition 2.

Probabilistic Systems with Nondeterminism

Now let us consider the behaviour of some nondeterministic-probabilistic cyclic system *M* in detail. We can assume that the initial system state *s* belongs to the set of operational states *S_{op}*. After its first iteration, the system nondeterministically chooses some distribution Δ_s from $\mathcal{PL}_M.s$ and then, according to this distribution, reaches some state *s'* with the non-zero probability $\Delta_s.s'$. At this point, if *s' ∈ S_τ*, the system terminates. Otherwise, the system starts a new iteration. It is easy to see that the

behavioural semantics of a nondeterministic-probabilistic cyclic system in Event-B is defined by a Markov decision process with the absorbing set S_τ [Put05, Whi93].

Nondeterminism has the demonic nature in Event-B, i.e., we do not have any control or information about which branch of execution will be chosen [BvW98]. Therefore, to reason about such a model, we have to consider the worst case scenario, i.e. the scenario that leads the system to terminate with the highest probability.

Definition 3 [Refinement for nondeterministic-probabilistic systems]. *For two nondeterministic-probabilistic Event-B models M and M' of cyclic systems such that $M \triangleq (\text{Initialisation}; \mathbf{do} J \rightarrow E \mathbf{od})$ and $M' \triangleq (\text{Initialisation}'; \mathbf{do} J' \rightarrow E' \mathbf{od})$, we say that M' is a refinement of M , if and only if*

1. M' is an Event-B refinement of M ($M \sqsubseteq M'$);
2. $\forall t \in \mathbb{N}_1 \cdot \mathbf{P}_{\min}\{\Box^{\leq t} J\} \leq \mathbf{P}_{\min}\{\Box^{\leq t} J'\}$,

where $\mathbf{P}_{\min}\{\Box^{\leq t} J\}$ is the minimum probability that J remains true during the first t iterations.

Remark 2. If the second refinement condition of the Definition 3 holds not for all t , but for some interval $t \in 1..T$, $T \in \mathbb{N}_1$, we say that M' is a *partial* refinement of M for $t \leq T$.

To evaluate the worst case reliability for probabilistic systems with non-determinism, we have to calculate the minimum probability $\mathbf{P}_{\min}\{\Box^{\leq t} J\}$. Let us assume that some cyclic system M is in a state s , while $\mathcal{PL}_M$ maps s to the *finite* set of distributions $\bar{\Delta}_s = \{\Delta_s^1, \Delta_s^2, \dots\}$. If we want to evaluate the worst case reliability of the system for this iteration, we just have to choose the distribution that maps s to the set of operational states with a minimal probability, i.e., the probability $\min_{\Delta \in \bar{\Delta}_s} \sum_{s' \in S_{op}} \Delta.s'$.

However, when the goal is to evaluate the worst case stochastic behaviour of the system within a time interval $[0, t]$, where $t \geq 1$, the calculation process of the resulting minimal probability becomes more complex. Because of the intricate nature of demonic nondeterminism, we cannot simply rely on the calculated fixed minimal probability for t iterations when calculating it for $t + 1$ iterations. The “demon” does not have to stick to its previous choices, so the minimal probability has to be re-calculated anew, now for $t + 1$ iterations.

Let us define the worst case reliability as a function $r(t, s)$, the arguments of which are the number of system iterations t and some initial state s .¹ The

¹For now, we assume that the initial system state s is deterministically defined. Later

function returns the minimal value of the reliability function $R(t)$ over all possible state distributions.

The definition of $r(t, s)$ is recursive. Two basic cases define the function values for respectively terminating (absorbing) states and the zero number of iterations:

$$\begin{aligned} s \in S_\tau &\Rightarrow \forall t \in \mathbb{N} \cdot r(t, s) = 0, \\ s \in S_{op} &\Rightarrow r(0, s) = 1. \end{aligned}$$

Generally, for $s \in S_{op}$, we can recursively define the function $r(t, s)$ in the following way:

$$\forall t \in \mathbb{N}_1, s \in S_{op} \cdot r(t, s) = \min_{\Delta \in \Delta_s} \sum_{s' \in S_{op}} \Delta.s' \cdot r(t-1, s').$$

Please note that the recursive function application essentially traverses all the possible operational state transitions and, based on that, operational state distributions, and then finds the minimal probability of the system staying operational.

Such an approach for defining an “absorbing” function is rather straightforward and widely used in the works based on Markov decision processes with absorbing sets (see [HW03] for instance).

Now we are ready to revisit our definition of reliability refinement for nondeterministic-probabilistic cyclic systems. Let us define, for such a system M , column-vector r_M^t with the elements $r_M^t(s) = r(t, s)$. Now, assuming that the initial state of the system is not defined deterministically but given instead by some initial state distribution, we can formulate the following proposition:

Proposition 2. *Assume that the initial system state is defined according to some probability distribution. Then for two nondeterministic-probabilistic Event-B models M and M' such that $M \hat{=} (\text{Initialisation}; \mathbf{do} J \rightarrow E \mathbf{od})$ and $M' \hat{=} (\text{Initialisation}'; \mathbf{do} J' \rightarrow E' \mathbf{od})$, the inequality*

$$\forall t \in \mathbb{N}_1 \cdot \mathbf{P}\{\Box^{\leq t} J\} \leq \mathbf{P}\{\Box^{\leq t} J'\}$$

is equivalent to

$$\forall t \in \mathbb{N}_1 \cdot [\Delta_0] \cdot r_M^t \leq [\Delta'_0] \cdot r_{M'}^t, \quad (10.30)$$

where $[\Delta_0]$ and $[\Delta'_0]$ are the initial state distribution row-vectors for the systems M and M' respectively.

we consider more general cases when s is given by some initial probability distribution or nondeterministically.

Proof Directly follows from our definition of r_M^t . ■

We can also assume that we do not have precise information about the system initial state, i.e., the initialisation takes the following form: $s \in S_0$, where $S_0 \subseteq S_{op}$. In this case we can define the refinement relation (10.30) as follows.

Remark 3. Assume that the initial system state is defined nondeterministically. Then the reliability refinement condition (10.30) is equivalent to

$$\forall t \in \mathbb{N}_1 \cdot \min_{s \in S_0} r(t, s) \leq \min_{s \in S'_0} r(t, s), \quad (10.31)$$

where S_0 and S'_0 are the sets of possible initial states for the systems M and M' respectively.

Reliability Refinement Verification

For fully probabilistic systems, we can often reduce the state space size using the lumping technique [KS60] or equally probabilistic bisimulation [LS91]. For nondeterministic probabilistic systems, a number of bisimulation techniques [Han95, SL95] have been also developed.

For simple system models, deriving the set of state distributions $\bar{S}$ and calculating reliability probabilities P_M^t for each refinement step can be done manually. However, for complex real-size systems this process can be extremely time and effort consuming. Therefore, it is beneficial to have an automatic tool support for routine calculations. Development and verification of Event-B models is supported by the Rodin platform [Rodb] – integrated extensible development environment for Event-B. However, at the moment the support for quantitative verification is sorely missing. To prove probabilistic refinement of Event-B models according to Definition 2 and Definition 3, we need to extend the Rodin platform with a dedicated plug-in or integrate some external tool.

One of the available automated techniques widely used for analysing systems that exhibit probabilistic behaviour is probabilistic model checking [BK08, Kwi07]. In particular, the probabilistic model checking frameworks like PRISM or MRMC [PRI, MRM] provide good tool support for formal modelling and verification of discrete- and continuous-time Markov processes. To enable the quantitative reliability analysis of Event-B models, it would be advantageous to develop a Rodin plug-in enabling automatic translation of Event-B models to existing probabilistic model checking frameworks.

10.2.4 Results Achieved

In this work we proposed an approach to integrating probabilistic assessment of reliability into Event-B modelling. We defined reliability of a cyclic system as the probability of the system to stay in its operational state for a given number of iterations. Our approach to augmenting Event-B models with probabilities allows us to give the semantic of a Markov process (or, in special cases, a Markov chain) to augmented models. In turn, this allow us to algebraically compute reliability by using any of numerous automated tools for reliability estimation.

10.3 Quantitative Verification of Safety

Safety is a property of a system to not endanger human life or environment [ALRL04]. To guarantee safety, designers employ various rigorous techniques for formal modelling and verification. Such techniques facilitate formal reasoning about system correctness. In particular, they allow us to guarantee that a safety invariant – a logical representation of safety – is always preserved during system execution. However, real safety-critical systems, i.e., the systems whose components are susceptible to various kinds of faults, are not “absolutely” safe. In other words, certain combinations of failures might lead to an occurrence of a hazard – a potentially dangerous situation that breaches safety requirements. While designing and certifying safety-critical systems, we should demonstrate that the probability of a hazard occurrence is acceptably low. In this work [TTL11a], we propose an approach to combining formal system modelling and quantitative safety analysis.

We have proposed design strategies that allow the developers to structure safety requirements according to the system abstraction layers. Essentially, such an approach can be seen as a process of extracting a fault tree – a logical representation of a hazardous situation in terms of the primitives used at different layers of abstraction. Eventually, we arrive at the representation of a hazardous situation in terms of failures of basic system components. After augmenting Event-B specifications with an explicit representation of probabilities of component failures [TTL10], we can use standard calculations to obtain a probabilistic evaluation of a hazard occurrence. As a result, we obtain an algebraic representation of probability of safety requirements violation. This probability is defined using the probabilities of system component failures. To illustrate our approach, we present a formal development and safety analysis of a radio-based railway crossing. We believe the proposed approach can potentially facilitate development, verification and assessment

of safety-critical systems.

10.3.1 General Overview of the Approach

In this work we focus on modelling of highly dynamic reactive control systems. Such systems provide instant control actions as a result of receiving stimuli from the controlled environment. Such a restriction prevents the system from executing automated error recovery, i.e. once a component fails, its failure is considered to be permanent and the system ceases its automatic functioning.

Generally, control systems are built in a layered fashion and reasoning about their behaviour is conducted by unfolding layers of abstraction. Deductive system safety analysis is performed in a similar way. We start by identifying a hazard – a dangerous undesirable situation associated with the system. By unfolding the layers of abstraction we formulate the hazard in terms of component states of different layers.

In an Event-B model, a hazard can be naturally defined as a predicate over the system variables. Sometimes, it is more convenient to reformulate a hazard as a dual safety requirement (property) that specifies a proper behaviour of a system in a hazardous situation. The general form of such a safety property is the following:

$$SAF \triangleq H(v) \Rightarrow K(v),$$

where the predicate $H(v)$ specifies a hazardous situation and the predicate $K(v)$ defines the safety requirements in the terms of the system variables and their states.

The essential properties of an Event-B model are usually formulated as invariants. However, to represent system behaviour realistically, our specification should include modelling of not only normal behaviour but also component failure occurrence. Since certain combinations of failures will lead to hazardous situations, we cannot guarantee “absolute” preservation of safety invariants. Indeed, the goal of development of safety-critical systems is to guarantee that the probability of violation of safety requirements is sufficiently small.

To assess the preservation of a desired safety property we will unfold it (in the refinement process) until it refers only to concrete system components that have direct impact on the system safety. To quantitatively evaluate this impact we require that these components are probabilistically modelled in Event-B using the available information about their reliability. Next we demonstrate how the process of unfolding the safety property from

the abstract to the required concrete representation can be integrated into the system development by refinement in Event-B.

Often, functioning of a system can be structured according to a number of *execution stages*. There is a specific component functionality associated with each stage. Since there is no possibility to replace or repair failed system components, we can divide the process of quantitative safety assessment into several consecutive steps, where each step corresponds to a particular stage of the system functioning. Moreover, a relationship between different failures of components and the system behaviour at a certain execution stage is preserved during all the subsequent stages. On the other hand, different subsystems can communicate with each other, which leads to possible additional dependencies between system failures (not necessarily within the same execution stage). This fact significantly complicates quantitative evaluation of the system safety.

We can unfold system safety properties either in a *backward* or in a *forward* way. In the backward unfolding we start from the last execution stage preceding the stage associated with the potentially hazardous situation. In the forward one we start from the first execution stage of the system and continue until the last stage just before the hazardous situation occurs. We follow the former approach. The main idea is to perform a stepwise analysis of any possible behaviour of all the subsystems at every execution stage preceding the hazardous situation, while gradually unfolding the abstract safety property in terms of new (concrete) variables representing faulty components of the system.

Specifically, in each refinement step, we have to establish the relationship between the newly introduced variables and the abstract variables present in the safety property. A standard way to achieve this is to formulate the required relationship as a number of safety invariants in Event-B. According to our development strategy, each such invariant establishes a connection between abstract and more concrete variables that have an impact on system safety. Moreover, the preservation of a safety invariant is usually verified for a particular subsystem at a specific stage. Therefore, we can define a general form of such an invariant in the following way:

$$I_s(v, u) \triangleq F(v) \Rightarrow (K(v) \Leftrightarrow L(u)),$$

where the predicate F restricts the execution stage and the subsystems involved, while the predicate $K \Leftrightarrow L$ relates the values of the newly introduced variables u with the values the abstract variables v present in the initially defined safety property or/and in the safety invariants defined in the previous refinement steps.

To calculate the probability of preservation of the safety property, the refinement process should be continued until all the abstract variables, used in the definition of the system safety property, are related to the concrete, probabilistically updated variables, representing various system failures or malfunctioning. The process of probability evaluation is rather straightforward and based on basic definitions and rules for calculating probabilities (see [Fel67] for instance).

Let us consider a small yet generic example illustrating the calculation of probability using Event-B safety invariants. We assume that the safety property SAF is defined as above. In addition, let us define two safety invariants – I_s and J_s – introduced in two subsequent refinement steps. More specifically,

$$I_s \hat{=} F \Rightarrow (K(v) \Leftrightarrow L_1(u_1) \vee L_2(u_2)) \text{ and } J_s \hat{=} \tilde{F} \Rightarrow (L_2(u_2) \Leftrightarrow N(w)),$$

where $u_1 \subset u, u_1 \neq \emptyset$ are updated probabilistically in the first refinement, while $u_2 = u \setminus u_1$ are still abstract in the first refinement machine and related by J_s to the probabilistically updated variables w in the following one. Let us note that the predicate $\tilde{F}$ must define the earlier stage of the system than the predicate F does. Then the probability that the safety property SAF is preserved by the system is

$$P_{SAF} = P\{K(v)\} = P\{L_1(u_1) \vee L_2(u_2)\} = P\{L_1(u_1) \vee N(w)\} = \\ P\{L_1(u_1)\} + P\{N(w)\} - P\{L_1(u_1) \wedge N(w)\},$$

where

$$P\{L_1(u_1) \wedge N(w)\} = P\{L_1(u_1)\} \cdot P\{N(w)\}$$

in the case of independent L_1 and N , and

$$P\{L_1(u_1) \wedge N(w)\} = P\{L_1(u_1)\} \cdot P\{N(w) \mid L_1(u_1)\}$$

otherwise. Note that the predicate $H(v)$ is not participating in the calculation of P_{SAF} directly. Instead, it defines “the time and the place” when and where the values of the variables u and v should be considered, and, as long as it specifies the hazardous situation following the stages defined by F and $\tilde{F}$, it can be understood as the *post-state* for all the probabilistic events.

Next we will demonstrate the approach presented above by a case study – an automatic railway crossing system.

10.3.2 Case Study

To illustrate safety analysis in the probabilistically enriched Event-B method, in this section we present a quantitative safety analysis of a radio-based

railway crossing. This case study is included into priority program 1064 of the German Research Council (DFG) prepared in cooperation with Deutsche Bahn AG. The main difference between the proposed technology and traditional control systems of railway crossings is that signals and sensors on the route are replaced by radio communication and software computations performed at the train and railway crossings. Formal system modelling of such a system has been undertaken previously [OS07, ORS07]. However, the presented methodology is focused on logical (qualitative) reasoning about safety and does not include quantitative safety analysis. Below we demonstrate how to integrate formal modelling and probabilistic safety analysis.

Let us now briefly describe the functioning of a radio-based railway crossing system. The train on the route continuously computes its position. When it approaches a crossing, it broadcasts a *close* request to the crossing. When the railway crossing receives the command *close*, it performs some routine control to ensure safe train passage. It includes switching on the traffic lights, that is followed by an attempt to close the barriers. Shortly before the train reaches the *latest braking point*, i.e., the latest point where it is still possible for the train to stop safely, it requests the *status* of the railway crossing. If the crossing is secured, it responds with a *release* signal, which indicates that the train may pass the crossing. Otherwise, the train has to brake and stop before the crossing. More detailed requirements can be found in [OS07] for instance.

In our development we abstract away from modelling train movement, calculating train positions and routine control by the railway crossing. Let us note that, any time when the train approaches to the railway crossing, it sequentially performs a number of predefined operations:

- it sends the *close* request to the crossing controller;
- after a delay it sends the *status* request;
- it awaits for an answer from the crossing controller.

The crossing controller, upon receiving the *close* request, tries to close the barriers and, if successful, sends the *release* signal to the train. Otherwise, it does not send any signal and in this case the train activates the emergency brakes. Our safety analysis focused on defining the hazardous events that may happen in such a railway crossing system due to different hardware and/or communication failures, and assess the probability of the hazard occurrences. We make the following fault assumptions:

- the radio communication is unreliable and can cause messages to be lost;

- the crossing barrier motors may fail to start;
- the positioning sensors that are used by the crossing controller to determine a physical position of the barriers are unreliable;
- the train emergency brakes may fail.

The abstract model. We start our development with identification of all the high-level subsystems we have to model. Essentially, our system consists of two main components – the train and the crossing controller. The system environment is represented by the physical position of the train. Therefore, each control cycle consists of three main phases – *Env*, *Train* and *Crossing*. To indicate the current *phase* the eponymous variable is used.

The type modelling abstract train positions is defined as the enumerated set of non-negative integers $POS\_SET = \{0, CRP, SRP, SRS, DS\}$, where $0 < CRP < SRP < SRS < DS$. Each value of POS_SET represents a specific position of the train. Here 0 stands for some initial train position outside the communication area, *CRP* and *SRP* stand for the close and status request points, and *SRS* and *DS* represent the safe reaction and danger spots respectively. The actual train position is modelled by the variable $train\_pos \in POS\_SET$. In addition, we use the boolean variable *emrg_brakes* to model the status of the train emergency brakes. We assume that initially they are not triggered, i.e., $emrg\_brakes = FALSE$.

The crossing has two barriers – one at each side of the crossing. The status of the barriers is modelled by the variables bar_1 and bar_2 that can take values *Opened* and *Closed*. We assume that both barriers are initially open.

The initial abstract machine *RailwayCrossing* is illustrated in Figure 10.19. We omit showing here the *Initialisation* event and the **Invariants** clause (it merely defines the types of variables). Due to lack of space, in the rest of the section we will also present only some selected excerpts of the model. The full Event-B specifications of the *Railway crossing system* can be found in [TTL11b].

In the machine *RailwayCrossing* we consider only the basic functionality of the system. Two events $UpdatePosition_1$ and $UpdatePosition_2$ are used to abstractly model train movement. The first event models the train movement outside the danger spot by updating the train abstract position according to the next value of the POS_SET . The event $UpdatePosition_2$ models the train behaviour after it has passed the last braking point or when it has stopped in the safe reaction spot. Essentially, this event represents the system termination (both safe and unsafe cases), which is modelled as

```

Machine RailwayCrossing
Variables train_pos, phase, emrg_brakes, bar1, bar2
Invariants ...
Events ...
  UpdatePosition1  $\hat{=}$ 
    when
      phase = Env  $\wedge$  train_pos < DS  $\wedge$  emrg_brakes = FALSE
    then
      train_pos := min( $\{p \mid p \in POS\_SET \wedge p > train\_pos\}$ ) || phase := Train
    end
  UpdatePosition2  $\hat{=}$ 
    when
      phase = Env  $\wedge$  ((train_pos = DS  $\wedge$  emrg_brakes = FALSE)  $\vee$  emrg_brakes =
    TRUE)
    then
      skip
    end
  TrainIdle  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos  $\neq$  SRS
    then
      phase := Crossing
    end
  TrainReact  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos = SRS
    then
      emrg_brakes : $\in$  BOOL || phase := Crossing
    end
  CrossingBars  $\hat{=}$ 
    when
      phase = Crossing  $\wedge$  train_pos = CRP
    then
      bar1, bar2 : $\in$  BAR_POS || phase := Env
    end
  CrossingIdle  $\hat{=}$ 
    when
      phase = Crossing  $\wedge$  train_pos  $\neq$  CRP
    then
      phase := Env
    end

```

Figure 10.19: Railway crossing: the abstract machine

infinite stuttering, i.e., keeping the system in the final state forever. Such an approach for modelling of the train movement is sufficient since we only analyse system behaviour within the train-crossing communication area, i.e., the area that consists of the close and status request points, and the safe reaction spot. A more realistic approach for modelling of the train movement is out of the scope of our safety analysis.

For the crossing controller, we abstractly model closing of the barriers by the event *CrossingBar*, which non-deterministically assigns the variables *bar₁* and *bar₂* from the set *BAR_POS*. Let us note that in the abstract

machine the crossing controller immediately knows when the train enters the close request area and makes an attempt to close the barriers. In further refinement steps we eliminate this unrealistic abstraction by introducing communication between the train and the crossing controller. In addition, in the *Train* phase the event *TrainReact* non-deterministically models triggering of the train emergency brakes in the safe reaction spot.

The hazard present in the system is the situation when the train passes the crossing while at least one barrier is not closed. In terms of the introduced system variables and their states it can be defined as follows:

$$train\_pos = DS \wedge (bar_1 = Opened \vee bar_2 = Opened).$$

In a more traditional (for Event-B invariants) form, this hazard can be dually reformulated as the following safety property:

$$train\_pos = SRS \wedge phase = Crossing \Rightarrow \\ (bar_1 = Closed \wedge bar_2 = Closed) \vee emrg\_brakes = TRUE. \quad (10.32)$$

This safety requirement can be interpreted as follows: after the train, being in the safe reaction spot, has reacted on signals from the crossing controller, the system is in the safe state only when both barriers are closed or the emergency brakes are activated. Obviously, this property cannot be formulated as an Event-B invariant – it might be violated due to possible communication and/or hardware failures. Our goal is to assess the probability of violation (or preservation) of the safety property (10.32). To achieve this, during the refinement process, we have to unfold (10.32) by introducing into the specification the representation of all the system components that have an impact on the system safety. Moreover, we should establish a relationship between the variables representing these components and the abstract variables presented in (10.32).

The first refinement. In the first refinement step we examine in detail the system behaviour at the safe reaction spot – the last train position preceding the danger spot where the hazard may occur. As a result, the abstract event *TrainReact* is refined by three events *TrainRelease₁*, *TrainRelease₂* and *TrainStop* that represent reaction of the train on the presence or absence of the release signal from the crossing controller. The first two events are used to model the situations when the release signal has been successfully delivered or lost respectively. The last one models the situation when the release signal has not been sent due to some problems at the crossing controller side. Please note that since the events *TrainRelease₂* and *TrainStop* perform the same actions, i.e., trigger the emergency brakes, they differ only in their guards.

The event *CrossingStatusReq* that “decides” whether to send or not to send the release signal is very abstract at this stage – it does not have any specific guards except those that define the system phase and train position. Moreover, the variable *release_snd* is updated in the event body non-deterministically. To model the failures of communication and emergency brakes, we introduce two new events with *probabilistic* bodies – the events *ReleaseComm* and *TrainDec* correspondingly. For convenience, we consider communication as a part of the receiving side behaviour. Thus the release communication failure occurrence is modelled in the *Train* phase while the train being in the *SRS* position. Some key details of the Event-B machine *RailwayCrossing_R1* that refines the abstract machine *RailwayCrossing* are shown in Figure 10.20.

The presence of concrete variables representing unreliable system components in *RailwayCrossing_R1* allows us to formulate two safety invariants (*saf_inv₁* and *saf_inv₂*) that glue the abstract variable *emrg_brakes* participating in the safety requirement (10.32) with the (more) concrete variables *release_rcv*, *emrg_brakes_failure*, *release_snd* and *release_com_failure*.

$$\text{saf\_inv}_1 : \text{train\_pos} = \text{SRS} \wedge \text{phase} = \text{Crossing} \Rightarrow (\text{emrg\_brakes} = \text{TRUE} \Leftrightarrow \text{release\_rcv} = \text{FALSE} \wedge \text{emrg\_brakes\_failure} = \text{FALSE})$$

$$\text{saf\_inv}_2 : \text{train\_pos} = \text{SRS} \wedge \text{phase} = \text{Crossing} \Rightarrow (\text{release\_rcv} = \text{FALSE} \Leftrightarrow \text{release\_snd} = \text{FALSE} \vee \text{release\_comm\_failure} = \text{TRUE})$$

We split the relationship between the variables into two invariant properties just to improve the readability and make the invariants easier to understand. Obviously, since the antecedents of both invariants coincide, one can easily merge them together by replacing the variable *release_rcv* in *saf_inv₁* with the right hand side of the equivalence in the consequent of *saf_inv₁*. Please note that the variable *release_snd* corresponds to a certain combination of system actions and hence should be further unfolded during the refinement process.

The second refinement. In the second refinement step we further elaborate on the system functionality. In particular, we model the request messages that the train sends to the crossing controller, as well as sensors that

```

Machine RailwayCrossing_R1
Variables ... , release_snd, release_rcv, emrg_brakes_failure, release_comm_failure, ...
Invariants ...
Events ...
  TrainRelease1  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos = SRS  $\wedge$  release_snd = TRUE
      release_comm_failure = FALSE  $\wedge$  deceleration = FALSE  $\wedge$  comm_ct = FALSE
    then
      emrg_brakes := FALSE || release_rcv := TRUE || phase := Crossing
    end
  TrainRelease2  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos = SRS  $\wedge$  release_snd = TRUE
      release_comm_failure = TRUE  $\wedge$  deceleration = FALSE  $\wedge$  comm_ct = FALSE
    then
      emrg_brakes :| emrg_brakes'  $\in$  BOOL  $\wedge$  (emrg_brakes' = TRUE  $\Leftrightarrow$ 
        emrg_brakes_failure = FALSE)
      release_rcv := TRUE || phase := Crossing
    end
  TrainStop  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos = SRS  $\wedge$  release_snd = FALSE  $\wedge$  deceleration =
      FALSE
    then
      ...
    end
  CrossingStatusReq  $\hat{=}$ 
    when
      phase = Crossing  $\wedge$  train_pos = SRP
    then
      release_snd : $\in$  BOOL || phase := Env
    end
  ReleaseComm  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos = SRS  $\wedge$  release_snd = TRUE  $\wedge$  comm_ct = TRUE
    then
      release_comm_failure  $\oplus$  | TRUE @ p1; FALSE @ 1 - p1 || comm_ct := FALSE
    end
  TrainDec  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos = SRS  $\wedge$  deceleration = TRUE
    then
      emrg_brakes_failure  $\oplus$  | TRUE @ p4; FALSE @ 1 - p4 || deceleration := FALSE
    end

```

Figure 10.20: Railway crossing: first refinement

read the position of the barriers. Selected excerpts from the second refinement machine *RailwayCrossing_R2* are shown in Figure 10.21. To model sending of the close and status requests by the train, we refine the event *TrainIdle* by two simple events *TrainCloseReq* and *TrainStatusReq* that activate sending of the close and status requests at the corresponding stages. In the crossing controller part, we refine the event *CrossingBars* by the event *CrossingCloseReq* that sets the actuators closing the barriers in response to

the close request from the train. Clearly, in the case of communication failure occurrence during the close request transmission, both barriers remain open.

```

Machine RailwayCrossing_R2
Variables ... , close_snd, close_rcv, status_snd, status_rcv,
               close_com_failure, status_com_failure, sensor1, sensor2 ...
Invariants ...
Events ...
  TrainCloseReq  $\hat{=}$ 
    when
      phase = Train  $\wedge$  train_pos = CRP
    then
      close_req_snd := TRUE || phase := Crossing
    end
    ...
  CrossingCloseReq  $\hat{=}$ 
    when
      phase = Crossing  $\wedge$  close_req_snd = TRUE  $\wedge$  comm_tc = FALSE
    then
      bar1, bar2 :| bar'1  $\in$  BAR_POS  $\wedge$  bar'2  $\in$  BAR_POS  $\wedge$ 
        (close_comm_failure = TRUE  $\Rightarrow$  bar'1 = Opened  $\wedge$  bar'2 = Opened)
      close_req_rcv :| close_req_rcv'  $\in$  BOOL  $\wedge$  (close_req_rcv' = TRUE  $\Leftrightarrow$ 
        close_comm_failure = FALSE)
      comm_tc := TRUE || phase := Env
    end
  CrossingStatusReq1  $\hat{=}$ 
    when
      phase = Crossing  $\wedge$  status_req_snd = TRUE  $\wedge$  close_req_rcv = TRUE  $\wedge$ 
      sens_reading = FALSE  $\wedge$  comm_tc = FALSE
    then
      release_snd :| release_snd'  $\in$  BOOL  $\wedge$  (release_snd' = TRUE  $\Leftrightarrow$ 
        status_comm_failure = FALSE  $\wedge$  sensor1 = Closed  $\wedge$  sensor2 = Closed)
      status_req_rcv :| status_req_rcv'  $\in$  BOOL  $\wedge$  (status_req_rcv' = TRUE  $\Leftrightarrow$ 
        status_comm_failure = FALSE)
      comm_tc := TRUE || phase := Env
    end
    ...
  ReadSensors  $\hat{=}$ 
    when
      phase = Crossing  $\wedge$  status_req_snd = TRUE  $\wedge$  sens_reading = TRUE
    then
      sensor1 := {bar1, bnot(bar1)} || sensor2 := {bar2, bnot(bar2)} || sens_reading :=
      FALSE
    end

```

Figure 10.21: Railway crossing: second refinement

Moreover, the abstract event *CrossingStatusReq* is refined by two events *CrossingStatusReq₁* and *CrossingStatusReq₂* to model a reaction of the crossing controller on the status request. The former event is used to model the situation when the close request has been successfully received (at the previous stage) and the latter one models the opposite situation. Notice that in the refined event *CrossingStatusReq₁* the controller sends the release signal only when it has received both request signals and identified that both barriers are closed. This interconnection is reflected in the safety invariant

saf_inv_3 .

$$\begin{aligned} saf\_inv_3 : train\_pos = SRP \wedge phase = Env \Rightarrow \\ (release\_snd = TRUE \Leftrightarrow close\_req\_rcv = TRUE \wedge \\ status\_req\_rcv = TRUE \wedge sensor_1 = Closed \wedge sensor_2 = Closed) \end{aligned}$$

Here the variables $sensor_1$ and $sensor_2$ represent values of the barrier positioning sensors. Let us remind that the sensors are unreliable and can return the actual position of the barriers incorrectly. Specifically, the sensors can get stuck at their previous values or spontaneously change the values to the opposite ones. In addition, to model the communication failures, we add two new events *CloseComm* and *StatusComm*. These events are similar to the *ReleaseComm* event of the *RailwayCrossing_R1* machine. Rather intuitive dependencies between the train requests delivery and communication failure occurrences are defined by a pair of safety invariants saf_inv_4 and saf_inv_5 .

$$\begin{aligned} saf\_inv_4 : train\_pos = SRP \wedge phase = Env \Rightarrow \\ (status\_req\_rcv = TRUE \Leftrightarrow status\_com\_failure = FALSE) \end{aligned}$$

$$\begin{aligned} saf\_inv_5 : train\_pos = CRP \wedge phase = Env \Rightarrow \\ (close\_req\_rcv = TRUE \Leftrightarrow close\_com\_failure = FALSE) \end{aligned}$$

The third refinement. In the third Event-B machine *RailwayCrossing_R3*, we refine the remaining abstract representation of components mentioned in the safety requirement (10.32), i.e., modelling of the barrier motors and positioning sensors. We introduce the new variables $bar\_failure_1$, $bar\_failure_2$, $sensor\_failure_1$ and $sensor\_failure_2$ to model the hardware failures. These variables are assigned probabilistically in the newly introduced events *BarStatus* and *SensorStatus* in the same way as it was done for the communication and emergency brakes failures in the first refinement. We refine *CrossingCloseReq* and *ReadSensors* events accordingly. Finally, we formulate four safety invariants saf_inv_6 , saf_inv_7 , saf_inv_8 and saf_inv_9 to specify the correlation between the physical position of the barriers, the sensor readings, and the hardware failures.

$$\begin{aligned} saf\_inv_6 : train\_pos = CRP \wedge phase = Env \Rightarrow (bar_1 = Closed \Leftrightarrow \\ bar\_failure_1 = FALSE \wedge close\_comm\_failure = FALSE) \end{aligned}$$

$$\text{saf\_inv}_7 : \text{train\_pos} = \text{CRP} \wedge \text{phase} = \text{Env} \Rightarrow (\text{bar}_2 = \text{Closed} \Leftrightarrow \\ \text{bar\_failure}_2 = \text{FALSE} \wedge \text{close\_comm\_failure} = \text{FALSE})$$

$$\text{saf\_inv}_8 : \text{train\_pos} = \text{SRP} \wedge \text{phase} = \text{Env} \Rightarrow (\text{sensor}_1 = \text{Closed} \Leftrightarrow \\ ((\text{bar}_1 = \text{Closed} \wedge \text{sensor\_failure}_1 = \text{FALSE}) \vee \\ (\text{bar}_1 = \text{Opened} \wedge \text{sensor\_failure}_1 = \text{TRUE})))$$

$$\text{saf\_inv}_9 : \text{train\_pos} = \text{SRP} \wedge \text{phase} = \text{Env} \Rightarrow (\text{sensor}_2 = \text{Closed} \Leftrightarrow \\ ((\text{bar}_2 = \text{Closed} \wedge \text{sensor\_failure}_2 = \text{FALSE}) \vee \\ (\text{bar}_2 = \text{Opened} \wedge \text{sensor\_failure}_2 = \text{TRUE})))$$

The first two invariants state that the crossing barrier can be closed (in the post-state) only when the controller has received the close request and the barrier motor has not failed to start. The second pair of invariants postulates that the positioning sensor may return the value *Closed* in two cases – when the barrier is closed and the sensor works properly, or when the barrier has got stuck while opened and the sensor misreads its position.

Once we have formulated the last four safety invariants, there is no longer any variable, in the safety property (10.32), that cannot be expressed via some probabilistically updated variables introduced during the refinement process. This allows us to calculate the probability P_{SAF} that (10.32) is preserved by the system:

$$P_{SAF} = P\{(\text{bar}_1 = \text{Closed} \wedge \text{bar}_2 = \text{Closed}) \vee \text{emrg\_brakes} = \text{TRUE}\} = \\ P\{\text{bar}_1 = \text{Closed} \wedge \text{bar}_2 = \text{Closed}\} + P\{\text{emrg\_brakes} = \text{TRUE}\} - \\ P\{\text{bar}_1 = \text{Closed} \wedge \text{bar}_2 = \text{Closed}\} \cdot \\ P\{\text{emrg\_brakes} = \text{TRUE} \mid \text{bar}_1 = \text{Closed} \wedge \text{bar}_2 = \text{Closed}\}.$$

Let us recall that we have identified four different types of failures in our system – the communication failure, the failure of the barrier motor, the sensor failure and emergency brakes failure. We suppose that the probabilities of all these failures are constant and equal to p_1 , p_2 , p_3 and p_4 correspondingly. The first probability presented in the sum above can be trivially calculated based on the safety invariants saf_inv_7 and saf_inv_8 :

$$P\{\text{bar}_1 = \text{Closed} \wedge \text{bar}_2 = \text{Closed}\} = \\ P\{\text{bar\_failure}_1 = \text{FALSE} \wedge \text{bar\_failure}_2 = \text{FALSE} \wedge \\ \text{close\_comm\_failure} = \text{FALSE}\} = (1 - p_1) \cdot (1 - p_2)^2.$$

Indeed, both barriers are closed only when the crossing controller received the close request and none of the barrier motors has failed. The calculation of the other two probabilities is slightly more complicated. Nevertheless, they can be straightforwardly obtained using all the safety invariants defined in the model and basic rules for calculating probability. We omit the computation details due to a lack of space. The resulting probability of preservation of the safety property (10.32) is:

$$P_{SAF} = (1 - p_1) \cdot (1 - p_2)^2 + \\ (1 - p_4) \cdot \left(1 - (1 - p_1)^3 \cdot (p_2 \cdot p_3 + (1 - p_2) \cdot (1 - p_3))^2 \right) - \\ (1 - p_1) \cdot (1 - p_2)^2 \cdot (1 - p_4) \cdot \left(1 - (1 - p_1)^2 \cdot (1 - p_3)^2 \right).$$

Please note that P_{SAF} is defined as a function of probabilities of component failures, i.e., probabilities $p_1, \dots, p_4$. Provided the numerical values of them are given, we can use the obtained formula to verify whether the system achieves the desired safety threshold.

10.3.3 Results Achieved

In this work we have proposed an approach to integrating quantitative safety assessment into formal system development in Event-B. The main merit of our approach is that of merging logical (qualitative) reasoning about correctness of system behaviour with probabilistic (quantitative) analysis of its safety. An application of our approach allows the designers to obtain a probability of hazard occurrence as a function over probabilities of component failures.

Essentially, our approach sets the guidelines for safety-explicit development in Event-B. We have shown how to explicitly define safety properties at different levels of refinement. The refinement process has facilitated not only correctness-preserving model transformations but also establishes a logical link between safety conditions at different levels of abstraction. It leads to deriving a logical representation of hazardous conditions. An explicit modelling of probabilities of component failures has allowed us to calculate the likelihood of hazard occurrence. The B Method and Event-B are successfully and intensively used in the development of safety-critical systems, particularly in the railway domain. We believe that our approach provides the developers with a promising solution unifying formal verification and quantitative reasoning.

In our future work we are planning to further extend the proposed approach to enable probabilistic safety assessment at the architectural level.

10.4 Continuous behaviour

Managing continuous behaviour in Event-B is clearly a major challenge, as Event-B is by nature event-based, and therefore discrete. A common argument for the sufficiency of discrete formalisms to describe continuous behaviour is that we only model observable events, and the smallest time granularity between these events can be fixed in advance. This approach is explored in Section 10.4.1 in the context of business information systems. An inadequacy of this approach is its incompatibility with refinement. The precision of observations increases as refinement proceeds, and this suggests that time granularity of observations should be allowed to decrease, but this is clearly incompatible with fixing this granularity in advance. In Section 10.4.2 we discuss early work in which an approach in which time is measured as the change in an observed entity, rather than as a fundamental dimension. This allows the granularity of observations to be decreased from one model to the next. In Section 10.4.3 we discuss a different approach again. In it the real-time analysis is separated from the Event-B modelling, and undertaken in the model checking tool Uppaal.

10.4.1 Time-bounded inconsistency

Maintaining consistency of data across distributed information systems is a significant problem, particularly for business-critical systems. One of the challenges is the fact that simplistic notions of consistency are often inadequate. A useful understanding of consistency must be application-dependant. For example, in a scenario in which a customer places an order, the quantity of a product in the sales order data should be the same in the final invoice and the dispatch note. The exact definition of consistency is application-dependent. For example, the goods in a delivery should be a subset of those ordered. We refer to this application-dependent consistency as *semantic consistency*.

Although semantic consistency is important in business information systems, inconsistency is also a fact of life, and processes and databases are often in temporarily inconsistent states. After a certain delay caused by late changes, manual steps, or recovery from errors, they must however assume a consistent state.

Modelling consistency and error recovery may help the developers to integrate late change or (partial) cancellation of business objects into business logics, and to mechanically analyse earlier in the development cycle the ways the business scenarios are constructed to recover from such errors and to adapt them if necessary before implementing them. Since reasoning about

semantic consistency involves the consideration of delays, handling consistency in models entails understanding and modelling time.

We have developed patterns for modelling and analysis of business processes that can accommodate time-bounded semantic inconsistency. The approach developed is presented in [BFRR10]. Time is not an explicit part of the Event-B language, and so the approach to modelling time is based on previous work on modelling time in Event-B [CMR07].

Patterns have been used in a number of ways within Event-B [BB09, HFA09a, Ili08b, CMR07]. We take the approach of [HFA09a] since our goal is to automatically apply the patterns we develop.

We therefore simplify the pattern presented in [CMR07] (the Time Constraint Pattern) and develop a pattern to add timing information to a model of a business information system. We then develop a further pattern for adding error recovery behaviour, and combine the two patterns developed.

We begin with a sender and a receiver over a channel, and wish to specify a time limit on the duration of messages in the channel. A direct application of Cansell's pattern results in a maximum duration for message transit, but excludes the possibility that messages may be delayed. We therefore generalise Cansell's pattern slightly, weakening the associated invariants but allowing delayed messages. The second pattern adds error recovery behaviour which allows us to treat messages differently depending on the duration of transit of the message. The patterns were applied to a Credit Request Process described in [PQZ08].

10.4.2 An alternative approach to adding time

We have begun work on an alternative approach to specifying time-dependent systems in Event-B. It is compatible with stepwise refinement within Event-B, in that in each machine in the model time is only considered to pass if an event occurs in the machine. Time is therefore not explicitly represented in the abstract models. This approach is closely related to the Leibnizian model of time [Fut08], in which time is not a fundamental dimension, but is used to distinguish the changes in an observed entity.

The approach is developed for state-transition systems in general, and specialised to Event-B. A timed Event-B system is built in the following way. Initially, an abstract (untimed) Event-B machine is built, in which all properties of interest which do not relate to timing aspects of the model may be specified and proved before we begin to consider the timing properties. The concrete (timed) model requires the description of the clock in the context. The timing requirements are expressed as (in)equalities between sets of the states in the concrete process. The concrete machine is developed with

reference to this clock.

In addition to the proof obligations which arise naturally, we require two properties to be shown of an system developed in the way, liveness and realisability. Liveness demonstrates that the system can always progress. It is violated if there is a combination of state and clock models that disagree on how the progress is made in the system so that some state transitions require impossible clock evolutions. At the verification level, this is manifested in a set of contradictory assumptions that make possible to discharge conditions that might not be otherwise provable. Realisability demonstrates that the system undergoes only finitely many transitions between any two time-points. A model that undergoes infinitely many transitions between any two time points cannot be realised in the physical world. In our context, such models are formed by coupling a non-terminating state model with a terminating clock model. In other words, the sum of durations of infinitely many transitions of a state model defines a valid time point value (assuming ∞ is not included in a clock alphabet). One reason for this could be that neither of transitions has a duration. A more interesting possibility is the so-called Zeno behaviour when durations of infinite transitions form convergent series.

10.4.3 Augmenting Event-B Modelling with Real-Time Verification

A large number of dependable embedded systems have stringent real-time requirements imposed on them. An analysis of real-time behaviour is usually conducted at the implementation level. However, it is desirable to obtain an evaluation of real-time properties early at the development cycle, i.e., at the modelling stage. Using data processing of BepiColombo SIXS/MIXS OBSW as a working example, Newcastle and Aabo have studied [ILT⁺11] possibilities to augment Event-B modelling with verification of real-time properties in the model checking tool Uppaal. In the approach, a process-based view is extracted from an Event-B model and translated into a timed automaton readable by Uppaal. The research has so far demonstrated that the approach is viable for the used example.

Essentially, the developed approach consists of three main stages: (1) "traditional" refinement in Event B; (2) defining an explicit concurrency model called a Process View (PV); (3) and finally, creating a timed-automata model and verification of the desired real-time constraints. Each part of this modelling chain has a specific purpose. The Event-B modelling automated by the Rodin platform facilitates reasoning about functional correctness of

the system under construction. A PV model explicitly defines processes and their synchronisation. It is a projection of an Event-B specification that allows us to represent the targeted system architecture and the corresponding communication infrastructure. Finally, a timed automata system model enables verification of the desired real-time properties.

In [ILT⁺11], we present the formal basis for creating such a modelling chain is presented and discuss the issues associated with its industrial use. The paper also defines the additional proof obligations needed to formally verify that a PV is a valid projection of an Event-B specification. Moreover, it demonstrates how to augment a PV model with clocks and timing constraints to arrive at a timed automata model. Indeed, the proof obligations verifying consistency between PV and Event-B models are amenable to the verification by any first order theorem prover, including the Rodin platform. Meanwhile, verification of liveness and desired real-time system properties relies on the widely-used model checker Uppaal.

Currently, the verification of system real-time properties is performed at the late development stages via simulation and testing with the hardware platform in the loop. Simulation and testing allow the designers to evaluate the actual real-time characteristics of the system, i.e., obtain the *absolute* values of WCET and other parameters. Our experiment has shown that it is rather unlikely that in the foreseeable future the existing techniques for formal real-time verification will acquire the same capabilities because the complexity of even rather small industrial systems is too overwhelming for real-time model checkers. On the other hand, such tools as Uppaal allow the designers quickly discover relative interdependencies between different characteristics of components, the impact of the degree of parallelism on the performance etc. Hence, they can greatly facilitate design space exploration.

An important lesson that we have quickly learnt in the course of our work was the importance of concern separation. To achieve a scalable solution to real-time verification, we had to abstract away the modelling details that are not relevant for the concurrency model. This motivated our research on the process view formalisation. We have also discovered that it would be quite impractical to try to automatically extract a concurrency model from an Event-B model. Such an approach would not allow the developers to capture the predefined architectural design solutions that should be implemented by the system. These observations have led us to creating a dedicated framework for constructing the concurrency model and defining the proof obligations that demonstrate consistency between Event-B and process view models.

While analysing the previous approaches to integrating the explicit reasoning about time into Event-B modelling, we have concluded that it is practical to introduce time straight into a PV model rather than into an Event-B

model. This results in a smooth transition into timed automata. Moreover, it allows us to avoid building a complex yet logically shallow model of a timed system in Event-B.

In our approach we have put a special emphasis on defining a set of well-formedness conditions ensuring that the abstractions are built in a sound way. To achieve a semantic anchoring between process view and Event-B models, we have formally expressed the verification conditions as theorems that can also be verified in the Rodin platform. As a future work, we are planning to restrict the axiomatisation of the Process View definition to the smallest possible kernel from which the rest of the conditions could be deduced. We also planning to experiment with deriving real-time concurrent system implementations by refinement and distilling the guidelines on the constructing and using a Process View model.

10.5 Probabilities

The main application area of formal engineering techniques and in particular Event-B is development and verification of dependable systems. Dependability is a multi-facet system characteristic that includes such attributes as safety, reliability, availability etc. [ALR01].

Safety is a property of a system to not endanger human life or environment [ALR01]. Formal modelling in Event-B allows us to guarantee that a safety invariant – a logical representation of safety – is always preserved during system execution. However, real safety-critical systems, i.e., the systems whose components are susceptible to various kinds of faults, are not “absolutely” safe. In other words, certain combinations of failures may lead to an occurrence of a hazard – a potentially dangerous situation breaching safety requirements. While designing and certifying safety-critical systems, we should demonstrate that the probability of a hazard occurrence is acceptably low. Hence Event-B modeling alone does not suffice and we need to integrate quantitative safety analysis into refinement process.

To achieve this, we rely on a probabilistic extension of Event-B [TTL10] and propose design strategies that allow the developers to structure safety requirements according to the system abstraction layers [TTL11c]. Essentially, such an approach can be seen as a process of extracting a fault tree – a logical representation of a hazardous situation in terms of the primitives used at different abstraction layers. Eventually, we arrive at the representation of a hazard in terms of failures of basic system components. Since our model explicitly contains probabilities of component failures, standard calculations allow us to obtain a probabilistic evaluation of a hazard occurrence.

As a result, we obtain an algebraic representation of the probability of safety violation. This probability is defined in terms of component failures.

Development of industrial-size dependable systems usually requires significant efforts and financial investments. While modeling in Event-B allows us to ensure functional correctness, it leaves aside non-functional requirements. However, if obtained design does not satisfy certain non-functional requirements, the costs of redevelopment might be very high. We have studied the problem of evaluating system responsiveness and reliability during the design process. Responsiveness is the likelihood that the system successfully completes service delivery within a certain time bound [TRF03]. In the design of safety-critical systems responsiveness might be tightly coupled with safety. Indeed, often we should guarantee that system reacts not only correctly but also promptly on the hazardous situation. Therefore, it is desirable to explore various design alternatives from the point of view of non-functional requirements early at the development process. This allows the developers to ensure that the chosen design satisfies dependability requirements. Furthermore, it helps to establish that a refined model is at least as reliable (responsive) as an abstract one. We proposed an approach to integrating reliability and responsiveness assessment into the refinement process.

Dealing with modelling of control systems in Event-B, we introduced the notion of operational guard. If we are interested in responsiveness then the operational guard defines the set of terminating states. In case we aim at evaluating reliability, the operational guard defines the states at which the system delivers services correctly. Both reliability and responsiveness are functions of time. Our notion of time is implicit – it is defined in terms of iterations of a control system. Our approach allows the designers to obtain a quantitative assessment of responsiveness and reliability.

Hallerstede and Hoang [HH07] have extended the Event-B framework with the qualitative probabilistic choice operator. This operator assigns new values to state variables with some positive but generally unknown probability. The proposed extension aims at introducing into Event-B the concept of *almost-certain convergence*. This extension is especially useful for proving termination.

The future work on probabilistic extension of Event-B aims at improving scalability of proposed techniques and providing an automated tool support. In particular, it would be fruitful to establish a link with probabilistic model checking techniques, e.g., PRISM or MRMC.

Chapter 11

Event-B and Choreography Models

V. Kozyura, A. Roth, W. Wei, S. Wieczorek

11.1 Introduction

Applying the concepts of Service-Oriented Architectures (SOA) has already become a mainstream in industry [Hef09]. The development of business applications according to these principles implies a layered design and implementation. In this chapter we describe an industrial approach to the verification of a consistency relation between such layers. In our case service choreographies defined by Message Choreography Models (MCM) and their corresponding implementation models represented as Business Objects are examined. By translating both into Event-B specifications we are able to prove the consistency relation between them. A number of case studies with realistic industrial software models were carried out which showed the solidness of our verification technique. Apart from giving details about our concrete realization, this chapter also discusses general challenges that have to be faced when developing a verification approach applicable for the real-world systems.

11.2 Checking Consistency Between Message Choreographies And Their Implementation Models

11.2.1 Overview

Service-oriented architectures (SOA) provide frameworks and methods to compose single services in order to realize complex business scenarios. At the lower end, a single service is described as a set of operations and message types. Its functions usually rely on a simple request-response pattern, which can be specified using standards like XML, SOAP, and WSDL. At the service integration level, more complicated specifications are needed to capture not only the formats of messages, but also the orders and dependencies among exchanged messages, in particular, both control-flow and data-flow dependencies. Thus, the challenge of the SOA-based development lies in the integration of different services according to the defined business processes. Choreography languages were introduced to describe the interaction protocols between service components communicating over message channels from the perspective of a global observer. Further, choreography languages also allow the specification of the communication behavior of local components.

SOA adoption is gaining pace toward becoming a mainstream [Hef09]. Being a leader in the area of business software, SAP delivers SOA via its service-enabled software (e.g., SAP Business ByDesign¹, SAP Business Suite²) and its open technology platform SAP NetWeaver³. In a model-driven fashion, the enterprise SOA developed at SAP utilizes many different types of models for business objects, deployment units, service components, service interfaces, integration scenarios, business process variants, and service choreographies [KP09].

The development of business applications according to the SOA principles implies a layered design and implementation. In [KRW09] we have shown how consistency can be verified for two layers of message choreography models: between global choreography models and local partner models. The work presented in Section 11.2 focuses on another consistency problem: the one between message choreographies (more precisely, local partner models) and their implementation models. Implementation models are not final implementation code. Even though very close to actual implementation details, they specify only the aspects relevant to the changes of internal life cycles

¹<http://www.sap.com/sme/solutions/businessmanagement/businessbydesign>

²<http://www.sap.com/solutions/business-suite>

³<http://www.sap.com/platform/netweaver>

of business objects in terms of state transition graphs. Therefore, our work is *not* concerned with source code analysis. Like our previous approach in [KRW09], we check consistencies through a translation into Event-B, a formal specification language supported by the Rodin platform [Abr10]. This work has been carried out in the context of SAP software developments, and we will report on the experience we gained from it.

Section 11.2.2 gives an overview of layered development and explains the motivation of ensuring consistency between different layers from an industrial perspective. Section 11.2.3 briefly discusses the choreography modeling language MCM and its verification. Section 11.2.4 introduces the basic concepts of implementation models used at SAP. Section 11.2.5 describes the transformation of implementation models to Event-B. How these formal representations are used for enforcing consistency and application specific properties is shown in Section 11.2.6. Section 11.2.7 discusses the lessons learnt.

11.2.2 Industrial Context

Our modeling approach is based on a three-layer architecture. In Section 11.2 we focus on the consistency relation between the last two layers. (1) **Global Choreography Models** (GCMs) describe a high-level view of the conversation between components. Based on labeled transition systems, they define every allowed sequence of messages as observed by a global observer. (2) **Local Partner Models** (LPMs) specify the communication-relevant behavior for each participating component. Each LPM has the same control structure as the GCM, and may have extra constraints on its local transitions. There is also a channel model (CM) describing the characteristics of the communication channels on which messages are exchanged between service components. (3) **Implementation Models** (IMs) are used as close abstractions of the final implementation code for business objects contained in local service components. They are described in terms of communicating UML state machines.

GCMs are used as a part of user requirements, and therefore we need to maintain the consistency between GCMs and IMs in order to guarantee that the implementation fulfills the requirements. There are various ways to define consistency relations between models [vG90]. Considering our application domain, we define consistency in terms of trace inclusion. Section 11.2 presents a formal approach to keep GCMs, LPMs and IMs consistent, by stepwise checkings between adjacent abstract layers as follows.

The **consistency between GCMs and LPMs** can be enforced by two approaches [DW07]: a generative approach where consistent LPMs are generated from GCMs, or a checking approach where GCMs and LPMs are

created separately and their consistency is afterward verified. While the first ensures that global and local views are always consistent, it makes changes to the local models considerably less flexible and more difficult. The latter approach allows for such “asymmetric” changes, but requires manual effort to update the global view when changes to the local models are made. In [KRW09] we described a mixed approach that takes best of both worlds.

The **consistency between LPMs and IMs** is the main concern of this section. We use the Event-B specification language and the Rodin platform [Abr10] to rigorously verify the consistency between LPMs and IMs. Consistencies are expressed in terms of event refinements in Event-B, and can be verified using either theorem proving or model checking (by ProB [LB08]).

Besides consistency, we also consider model specific properties such as absence of deadlocks and other safety properties. In order to check these properties, we either formulate them as additional invariants and prove their correctness, or express them in LTL formulas which are then validated by the ProB model checker.

11.2.3 Global Choreography and Local Partner Models

A message choreography model (MCM) [WKR⁺09, WRS⁺09, WRSC08] complements the static information of communication interfaces with dynamic information on message exchanging sequences and dependencies. Due to limited space, we are unable to give a detailed description of the MCM language. We use an example model from [WKR⁺09] to briefly introduce the modeling elements in MCM.

Two service components, a buyer and a seller, negotiate a sales order. The buyer starts the communication by sending a *Request* message that will be answered with a *Confirm* message by the seller. The buyer afterward has the choice of either to send a *Cancel* message that rolls back the previous communication and allows to restart the negotiation or to send an *Order* message that successfully concludes the ordering process. We assume a (reliable) communication channel that is not necessarily preserving the message order. Because of this a *Cancel* message can be delivered after a new negotiation process already started.

Figure 13.1 shows the MCM model for the above example, which consists of one GCM, two LPMs, and a channel model. In the GCM at the top of Figure 13.1, the arrows labeled with an envelope depict the interactions *Request*, *Confirm*, *Cancel*, *Order*, and *Cancel(deprecated)*⁴ which are ordered

⁴Deprecated here means that the message is out-dated and no-longer relevant as the

with the help of the states *Start*, *Request*, *Reserved*, and *Ordered*. The states connected with a filled circle, i.e. *Ordered* and *Start* are so-called target. Only in these states, the communication between the partners is allowed to terminate.

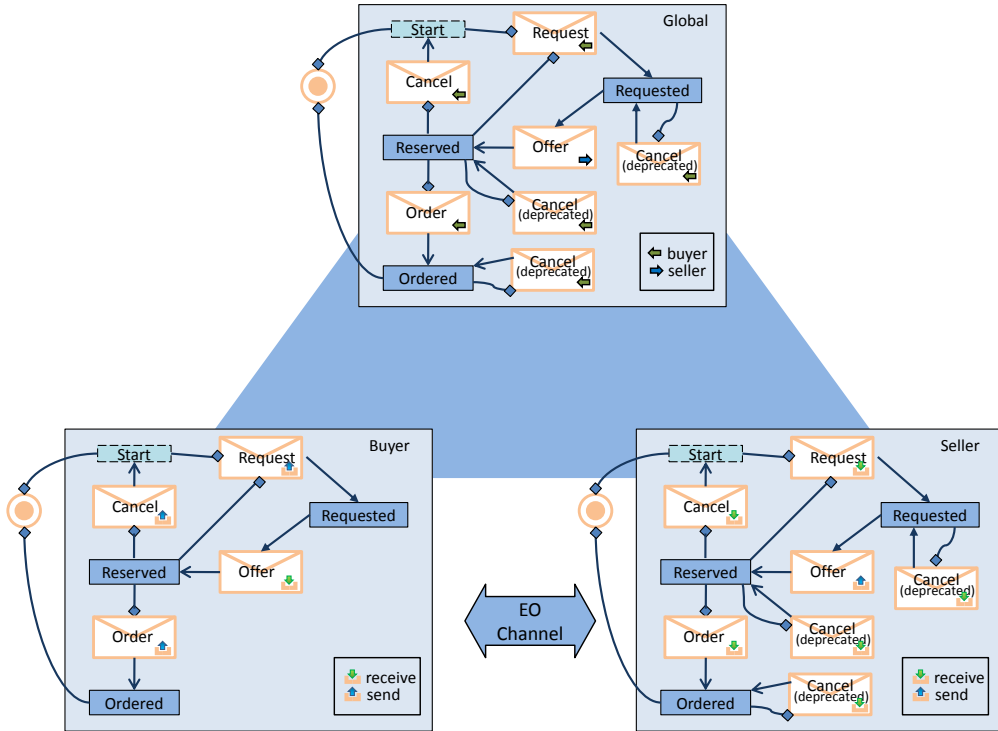


Figure 11.1: GCM (top) of the choreography and LPMs of the buyer (left) and the seller (right)

The LPM of the buyer partner of our example is depicted in the lower left part of Figure 13.1. It is a structural copy of the GCM, but the interaction symbols now represent send or receive events of the buyer. Moreover some send-events are "inhibited" by special local constraints. It is for example inhibited that a *Cancel(deprecated)* is ever sent (thus these send-events have been erased) and that a *Request* is sent in the *Reserved* state. However, due to possible message overtaking on a channel that does not guarantee to enforce the message order during transmission, receiving a deprecated *Cancel* is possible on the seller side. The LPM of the seller is depicted in the lower right part of Figure 13.1 with the exact structure as the GCM.

negotiation has been restarted.

MCM can be naturally translated into Event-B: interactions are simply represented as events, and the consistency between GCM and LPMs is expressed by Event-B refinement. The translation was already implemented, and also easily integrated with other tools, such as an MCM editor, thanks to the extensibility of the Eclipse-based Rodin platform. The details of the translation can be found in [WKR⁺09]. Here, we sketch the translation as follows.

For each transition in the GCM we generate exactly one event. For representing the states we define global status variables. In the local model we generate events representing sending and receiving of messages. Depending on the viewpoint either the send or the receive event can be defined to be a refinement of the corresponding interaction in GCM. The global status variable is duplicated for each LPM. In receive events, local variables (parameters) are used in order to obtain some message from a channel. A channel is defined as a global variable of type $\mathcal{P}(T)$, where T is a set of possible message types, denoting the set of messages being exchanged. It is initialized with $\emptyset$. Typically, we have two partners P_1 and P_2 and two sequencing contexts (exactly once (EO) and exactly once in order (EOIO)). In that case we obtain four possible channels in the model (two in each direction).

The purpose of the verification procedure is to prove local enforceability property for choreographies. In [WRS⁺09] we have defined a notion of local enforceability as a trace inclusion: Traces of the local model must be a subset of traces of the global model. Trace inclusion can be proved by showing that the local model is a refinement of the corresponding global one, with the help of the translation to Event-B.

In [KR09] it is shown how to generate automatically the gluing invariants between global and local models, which are for the practical examples usually enough in order to prove the refinement relation without adding any additional invariants.

11.2.4 Implementation Model

Business objects (BO) are basic units of business data and logic, which are contained in service components. Their life cycle states can be influenced by inter-BO communication described in choreography models. In this section, we model the life cycle of business objects as implementation models. These can be considered as refinements of their communication interfaces (i.e., local partner models), as illustrated in Figure 11.2. A business object contains a number of nodes organized in a tree-like structure. The changes made to each business node can be modeled using a UML State Diagram [OMG].

We define implementation models as model “templates” from which many

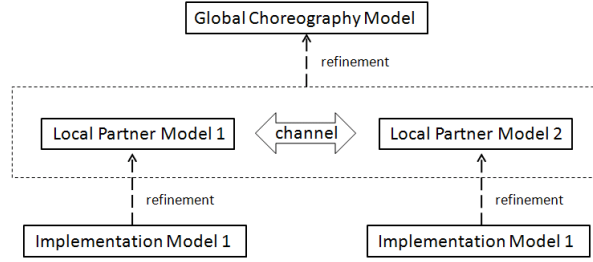


Figure 11.2: Refinement relations

model instances can be generated, which satisfy further constraints specified in templates. Due to limited space, we only give an intuitive introduction to implementation models using the example in Figure 11.3.

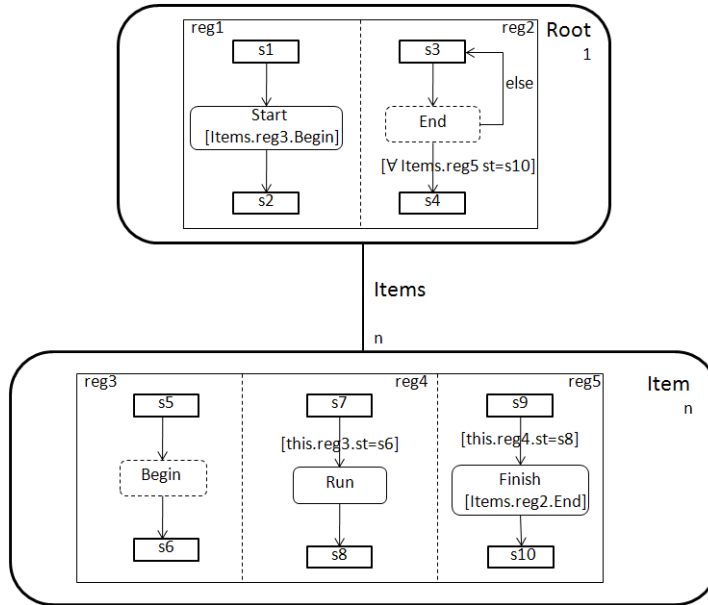


Figure 11.3: Example of an implementation model

An implementation model contains (1) a set of *node types* such as **Root** and **Item** in the example; (2) one single root node type (**Root** in our example); and (3) a set of node relation types such as **Items** that associates the root with a set of **Item** nodes. Furthermore, there are two sets of constraints: (1) The first set specifies how many nodes of each type are allowed for any BO instance, which is abbreviated by the number in the up-right corner of the corresponding node type. In our example, there can be only one **Root** node per BO instance (as indicated by the number 1) and an arbitrary number of

Item nodes per BO instance (as indicated by “**n**”). (2) The second constraint set specifies the multiplicity of each node relation type. In our example, the **Items** relation is a one-to-many mapping, i.e., an arbitrary number of **Items** can be associated with the root with respect to this relation.

Our variant of state machines is demanded by the great complexity of business process logic in the application context of the implementation models. Every node type has a state machine that contains a set of concurrent regions. Each region is an orthogonal part of the state machine that runs in parallel to other regions. Each region reflects the independent update of some system attribute. For example, the state machine of **Root** has two regions **reg1** and **reg2**. Each region has a set of states with one single initial state. Unlike traditional UML state machines, a transition in our variant state machine is no longer a simple pair of source and target state. First, a transition may be either *active* or *passive*. An active transition can be fired as long as its firing condition is satisfied. On the contrary, a passive transition can only be invoked by other transitions. Active transitions are graphically denoted as solid lines, and passive transitions are dotted lines. Second, the firing condition of a transition may be very complex in the sense that it may reference states across region boundaries or even node boundaries. Moreover, the effect of a transition may enforce the firing of transitions in other regions or in other state machines.

We need the following vocabulary to reference states and transitions of other state machines. Let *this* refer to the current node being considered. If n is a node, then $n.parent$ refers to the parent of n . If f is a relation type, then $n.f$ is the set of children nodes of n associated with n via the relation f . Moreover, $n.reg$ represents the region reg in n , whose current state is represented by $n.reg.st$. Finally, $n.reg.t$ refers to the transition t in reg .

A transition has a firing guard, a set of (pre_i, s_i) pairs that selects the next state s_i according to a certain pre-condition pre_i , and a sequence of transitions that must be invoked in order. In our example, **Start** is an active transition, whose guard is that the current state must be **s1**. When it is fired, the next state is **s2**, and it will enforce the transitions **Begin** in all **Item** nodes to be taken. The transition **End** is a passive transition, and can be fired only if it is invoked by the **Finish** in one of the **Item** nodes. It ends up in two possible next states: (1) If the regions **reg5** in all **Item** nodes are in state **s10**, then the next state is **s4**; (2) Otherwise, the next state is **s3**. The firing of **End** does not enforce any other transitions to be fired. Note that since the execution of a transition may invoke executions of other transitions, there is no guarantee for termination. One has to prove that the execution of any active transition indeed terminates.

11.2.5 Translation of Implementation Models to Event-B

In this section we describe a translation of IMs (node structures and state machines) into Event-B (For more information on EventB see [Abr10]). The translation is a challenging task because the translation should not only be sound, concise, and well-structured, but also allow as many properties to be automatically proved as possible. In Section 5.3 we show how the translation can be optimized in order to simplify Event-B representations and proofs and in Section 6.3 we compare results obtained by optimized and non-optimized translations.

Translation of model structure

We first show how to translate the tree structure of an implementation model into Event-B. The example in Figure 11.3 is a simplified version of the example in Figure 11.4. We consider the more complex version of the example in Figure 11.4 only in Section 11.2.5 in order to demonstrate some aspects of the translation of a tree-structure. In Section 11.2.5 and further we will continue with the simplified version in Figure 11.3.

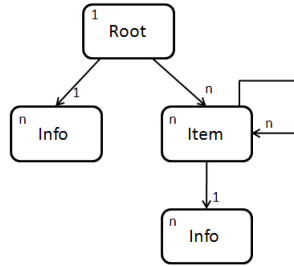
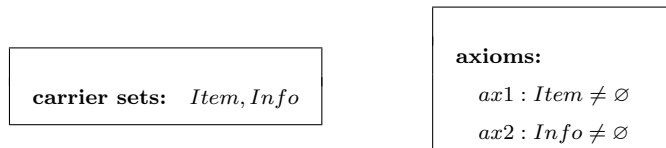


Figure 11.4: Example of a node structure

For a node type, if it may have only one node of this type per BO instance, such as the **Root** node in the example, there is no need to explicitly represent this node in Event-B. Otherwise, we use an abstract carrier set to denote all its node instances, which is the case for node types **Item** and **Info**. For these carrier sets, we need additional constraints stating that they are non-empty.



Given a node relation r that associates nodes of type t_1 with their children nodes of type t_2 , we need to distinguish the following two cases. In the first case, there is only one instance of type t_1 , say, the node n_1 . If r is a one-to-one mapping, we do not need to explicitly represent r since there can be only one node of type t_2 associated with its parent n_1 through r . Otherwise, if r is one-to-many, then we can represent r as the set of all children nodes of n_1 such that they are associated with n_1 through r . In the second case in which there are multiple nodes of type t_1 , we denote r using its inverse relation r^{-1} , which maps each node of type t_2 to its parent of type t_1 such that they are related by r . This is because r^{-1} is a function and results in simpler Event-B representations and proofs. When all parents of t_2 -nodes are of type t_1 , r^{-1} is a total function. Otherwise, it is a partial function. Moreover, if r is one-to-one, then r^{-1} is injective. The following shows how node relations in Figure 11.4 are translated in Event-B.

variables: $root\_info, root\_item, item\_item, item\_info$

invariants:

$inv1 : root\_info \in Info$

$inv2 : root\_item \in \mathcal{P}(Item)$

$inv3 : item\_item \in root\_item \text{ functional } Item \setminus root\_item$

$inv4 : item\_info \in Info \rightarrow Item$

In our example, an **Item** node can be associated with a child **Item** node, which we refer to as a *sub-item* (Sub-items do not have further sub-items) ⁵.

The following shows how variables denoting node relations are initialized. They are initialized non-deterministically in order to cover all possible model instances. There are also additional constraints: For example, the constraint $root\_item' \cap dom(item\_item') = \emptyset$ says that the set of items related to the root is disjoint from the set of items related to other items.

⁵Sometimes we may need sequences of item-item relations of arbitrary length. In this case, we can introduce the transitive closure, for example, we can reuse the definition of transitive closure from the Event-B mathematical toolkit

```

init
begin
  act1 : item_info, root_info :|
    item_info' ∈ Info  $\mapsto$  Item  $\wedge$  root_info' ∈ Info  $\wedge$ 
    root_info'  $\notin$  dom(item_info')  $\wedge$  root_info'  $\cup$  dom(item_info') = Info

  act2 : root_item, item_item :|
    root_item' ∈  $\mathcal{P}(\text{Item})$   $\wedge$  item_item' ∈ Item  $\nrightarrow$  Item  $\wedge$ 
    root_item'  $\cap$  dom(item_item') =  $\emptyset$   $\wedge$ 
    root_item'  $\cup$  dom(item_item') = Tasks
end

```

Translation of state machines

In this section we describe a translation of state machines, and show the necessity of introducing optimizations of the translation.

We define an abstract carrier set for the states of each region. For a node type that may have only one node instance, we use a variable for each region of the node to denote the current state of the region. For a node type with multiple instances, we define a function for each region that maps each node instance to the current state of the region in that particular node. As examples, the current state of **reg1** in Figure 11.3 is translated to a variable $reg1.st \in reg1.States$; and the current state of **reg3** is a function $reg3.st \in Items \rightarrow reg3.States$.

Now we show how transitions are translated. Let t be an active transition with a guard g and a set of pairs $(pre_1, s_1), \dots, (pre_n, s_n)$ defining the next state. Furthermore, t enforces a sequence of transitions $t_1, \dots, t_m$ to be taken. For simplicity reason, each enforced transition t_i has a guard **true**, a set of pairs $(pre_{i1}, s_{i1}), \dots, (pre_{ik_i}, s_{ik_i})$, and does not further enforce other transitions. The transition t is translated as follows:

```

EventT
  when
    grd1 : g
  then
    act1 : st, st1, ..., stm :|
      ((pre1 ∧ st' = s1) ∨ ... ∨ (pren ∧ st' = sn)) ∧
      ...
      ((prei1 ∧ st'i = si1) ∨ ... ∨ (preiki ∧ st'i = siki)) ∧
      ...
  end

```

In the above code we use st and $st_1, \dots, st_m$ to denote the current states of those regions that contain transitions t and $t_1, \dots, t_n$, for readability reason. Note that the effects of enforced transitions are specified together with the effect of the transition enforcing them in one Event-B event. This guarantees that the enforced transitions are indeed executed. Since passive transitions can only be invoked by others, their translations are always included in the Event-B events of some active transitions.

As an example, we show how transitions in Figure 11.3 are translated. The relation **Items** is represented as a subset of all **Item** nodes since there is only one **Root** node.

<pre> Start when grd1 : reg1.st = s1 then act1 : reg1.st := s2 act2 : reg3.st := Items × {s6} end </pre>	<pre> Run any node where grd1 : node ∈ Items grd2 : reg4.st(node) = s7 grd3 : reg3.st(node) = s6 then act1 : reg4.st(node) := s8 end </pre>


```

Finish
  any  node
  where
    grd1 : node ∈ Items
    grd2 : reg5.st(node) = s9
    grd3 : reg4.st(node) = s8
  then
    act1 : reg5.st, reg2.st : |
      (reg5.st'(node) = s10) ∧
      (∀ n ∈ Items ∧ n ≠ node ⇒ reg5.st'(n) = reg5.st(n)) ∧
      ( (∀ n ∈ Items ⇒ reg5.st'(n) = s10) ∧ reg2.st' = s4) ∨
      ((∃ n ∈ Items ⇒ reg5.st'(n) ≠ s10) ∧ reg2.st' = s3) )
  end

```

Note how complex the translation can be even for a relatively simple transition such as **Finish**, since the effect of the transition as well as the effects of all enforced transitions must be specified within one Event-B event. In particular, all effects of **Finish** are specified in one Event-B action **act1**. Unfortunately, **act1** cannot be broken into several smaller actions, because the next state of **reg2** depends on the next state of **reg5** in **Item** nodes. Such high complexity of the translation may significantly reduce the readability and provability of the translated model. Thus, in the next section we explore two possibilities of optimizing the translation of transitions.

Optimizations

Using implications in specifying preconditions of next states. A transition may have several potential next states, each depending on a certain precondition. The straightforward translation specifies the choice of the next state using disjunctions (see the previous section). However, this results in complex and less readable Event-B action, and also makes automated provers to become less effective. As a solution⁶, we may express the dependencies between preconditions and next states using implications in Event-B guards instead of using disjunctions in Event-B actions. This can be illustrated by the optimized translation of the transition **Finish** in Figure 11.3 as shown below. Two new variables **st5** and **st2** are introduced to express the dependencies of preconditions and next states for the regions

⁶We thank Michael Buttler for his suggestions and discussions for this optimization solution.

`reg5` and `reg2`. Their values are then used in the update of the next states of the regions.

```

Finish
  any  node, st5, st2
  where
    grd1 : node ∈ Items
    grd2 : reg5.st(node) = s9
    grd3 : reg4.st(node) = s8
    grd4 : st5 ∈ (Item → reg5.States)
    grd5 : st2 ∈ reg2.States
    grd6 : st5(node) = s10
    grd7 : ∀ n ∈ Items ∧ n ≠ node ⇒ st5(n) = reg5.st(n)
    grd8 : (∀ n ∈ Items ⇒ st5(n) = s10) ⇒ st2 = s4
    grd9 : (∃ n ∈ Items ⇒ st5(n) ≠ s10) ⇒ st2 = s3
  then
    act1 : reg5.st, reg2.st := st5, st2
  end

```

Using set operators. In the above model, there are a few guards containing quantifiers (see `grd7`, `grd8` and `grd9`). This may result in difficulties for automated provers to discharge proof obligations that make use of these guards as hypotheses. The reason is that the quantifiers in these guards need to be instantiated with concrete values during the proof, which requires the automated provers to make a choice in case several concrete values are available for instantiation. As a potential solution, we may consider to transform these guards into quantifier-free forms that use set operators. The advantage of using set operators is that the provers can now apply simplification rules for sets without facing choice of instantiations. As an example, the guard `grd7` can be rewritten as below, where $\triangleleft$ is domain subtraction:

$$grd7 : \{node\} \triangleleft st5 = \{node\} \triangleleft reg5.st$$

In a similar manner, the guards `grd8` and `grd9` can be written as

$$grd8 : ran(st5) = \{s10\} \Rightarrow st2 = s4$$

$$grd9 : st5 \triangleright \{s10\} \neq \emptyset \Rightarrow st2 = s3$$

In our experiments we did witness more automatically discharged proof obligations after eliminating quantifiers by set operators. However, using set operators is not always helpful, especially when an automated prover employs

such a proof strategy that translates set operators back to their quantifier-based versions. In future work we still need to assess how effective such quantifier elimination may improve the performance of automated provers, and which proof tactics should be used to better exploit the advantages of set operators. We will also design an automated procedure to translate guards to their quantifier free versions.

11.2.6 Analysis of Implementation Model

Using Event-B translations, we show how to check the consistency between local partner models and implementation models, and how to verify application specific properties for implementation models. We will also briefly describe how we conduct our experiments and discuss our experiences.

Checking Consistency Relation

The main purpose of our work is to check the consistency between message choreography models and their implementation models. As the consistency between GCM and LPMs can be verified [KR09], it suffices to show that each LPM is consistent with its implementation model. This means that all behavior of the IM can be also observed in the LPM, which corresponds to proving in Event-B that the IM is a refinement of the LPM.

In Event-B, the refinement between two machines is defined using gluing invariants that describe the relations between abstract variables and concrete variables. In [KR09] we presented an automated gluing invariant construction method for proving consistency between GCM and LPMs. For the time being, constructing gluing invariants for the consistency between LPMs and IMs is done manually with MCM tool support, using expert knowledge of specific models. A typical gluing invariant relates the states in a local partner model to the states in the corresponding implementation models. An example would be that, if the LPM is in state s , then some region in a state machine in the IM must be in one of the states $s_1, \dots, s_n$. Then, we have to prove that each transition in the IM results in a change of states that preserves the gluing invariants with respect to the change of states by the transition in the LPM which it refines.

The consistency between an LPM and its IM can be verified using either the ProB model checker [LB08] or the Atelier B provers [Abr10]. The advantage of model checking is that it is fully automated. However, it suffers from the state space explosion problem. As a solution, we may set bounds for integer variables and set sizes in ProB to reduce the explored portion of

the state space. This, however, cannot assure the consistency beyond the bounds that we set.

On the contrary, theorem proving does not need to explore the state space of a model. But it often requires human assistance to discharge proof obligations for large complex models. Besides, we may need to manually introduce auxiliary lemmas. For example, for the IM in Figure 11.3 we need the following additional invariant which says that if the root machine has not started then no items connected to the root has started their work.

$$(Root.reg1.st = st1) \Rightarrow (\forall item \in Items \Rightarrow item.reg3.st = s5).$$

Checking Application Specific Properties

We can verify application specific properties for implementation models such as deadlock freedom or other general safety and liveness properties. For example, we can check the following properties for the IM in Figure 11.3 expressing relations between states in the root node and in item nodes:

$$(Root.reg1.st = s1) \Rightarrow (\forall item \in Items \Rightarrow item.reg3.st = s5)$$

$$(\forall item \in Items \Rightarrow item.reg5.st = s10) \Rightarrow (Root.reg2.st = s4)$$

These properties can be expressed as invariants in the Event-B translation, and can be checked by either model checking or theorem proving. In ProB, we can also formulate and check LTL-expressible properties.

Experimental Results

We built an IM editor based on EMF (Eclipse Modeling Framework), and an automated translator from IMs to Event-B, in which translation optimization can be optionally enabled. The translation from LPMs to Event-B was already implemented in earlier work [WKR⁺09]. Using these tools, we conducted several case studies using real-life software models from the SAP ByDesign development environment. Due to confidentiality reasons, we are unable to disclose the details of the models that we use in experimentation.

We first used the ProB model checker to check both consistency and application specific properties. ProB is powerful enough to verify these models of considerable sizes, with the texts of some actions produced by the translation each spanning one or two full pages in print form. For a typical IM, its Event-B translation contains 25 events, and it took only 2 – 3 seconds for ProB to complete the checking.

Using the automated theorem provers for verification is only possible after applying the translation optimizations (see Sec. 11.2.5). The average number of proof obligations (PO) in our experiments was 150. Without optimization only a few of them could be proven automatically. After applying the optimization, 135 POs (90% of the total) were automatically discharged. We successfully proved consistency and checked certain application specific properties for all models.

The main difficulty here was the introduction of auxiliary invariants, which is an iterative process requiring expert knowledge of the models. The average number of invariants for one system was 27 with 11 type invariants, 7 gluing invariants and 9 auxiliary invariants describing the internal behavior of IMs. In the future we will design some automated procedures to assist in discovering auxiliary lemmas.

11.2.7 Summary

One advantage of layered designs is that assuring consistency between message choreographies and implementation models can be broken down into checking consistencies between adjacent layers. As our previous work examines the adherence between global choreography models and local partner models, this section shows how to check consistencies between local partner models and implementation models through automated translations into Event-B. We have also shown that application specific properties (e.g., deadlock freedom) can be verified at the level of implementation models. Practical evaluations with real-life models show a promising applicability of our approach on the industrial development of business applications.

Chapter 12

Structured Data and Mathematical Extensions

M. Butler, I. Maamria, A. Rezazadeh, M. Schmalz

12.1 Structured Data-Types and Refinement

As a part of formal modelling we may need to model structured data. A structured data-type can have multiple fields or elements which can be introduced at ones or gradually during successive refinement steps. The core mathematical language of Event-B currently does not support a syntax for the direct definition of structured types such as records or class structures. Nevertheless it is possible to model structured types using standard constructs provided in Event-B. Structured data-types then can be extended/enriched with extra fields in analogy with the general refinement approach. In this section we show how these can be achieved.

For example, suppose we wish to model a record structure R with fields $fld1$ and $fld2$ (with type $TP1$ and $TP2$ respectively). Let us use the following 'invented' syntax for this (based on VDM syntax but currently not part of Event-B syntax):

RECORD $R ::$ $/* \text{Record Type} */$
 $fld1 \in TP1 ,$ $/* \text{Field} */$
 $fld2 \in TP2 ;$ $/* \text{Field} */$

We can model this structure in Event-B by introducing (in a context) a carrier set R and two functions e and f as constants as follows:

SETS $R, TP1, TP2$ CONSTANTS $fld1, fld2$ AXIOMS $fld1 \in R \rightarrow TP1$ $fld2 \in R \rightarrow TP2$ END

The (constant) functions $fld1$ and $fld2$ are projection functions that can be used to extract the appropriate field values. That is, given an element $r \in R$ representing a record structure, we write $fld1(r)$ for the $fld1$ component of r and $fld2(r)$ for the $fld2$ component of r .

$TP1$ and $TP2$ can be any type definable in Event-B, including a type representing a record structure.

The approach to modelling structured types is based on a proposal by Evans and Butler [EB06]. We refer to this approach to modelling records as a projection-based approach. Later we will look at a constructor-based approach and make the link between the two approaches.

12.1.1 Constructing Structured Values

Suppose we have a variable v in a machine whose type is the structure R defined above:

INVARIANT $v \in R$

We wish to assign a structured value to v whose $fld1$ field has some value $e1$ and whose $fld2$ field has some value $f1$. This can be achieved by specifying the choice of an event parameter r whose fields are constrained by appropriate guards and assigning parameter r to the machine variable v . This is shown in the following event:

```

Ev1 =
    ANY  $r$  WHERE
         $r \in R$ 
         $fld1(r) = e1$ 
         $fld2(r) = f1$ 
    THEN
         $v := r$ 
    END

```

If we only wish to update some fields and leave others unchanged, this needs to be done by specifying explicitly that some fields remain unchanged. This is shown in the following example where only the e field is modified:

```

Ev1 =
    ANY  $r$  WHERE
         $r \in R$ 
         $fld1(r) = e1$ 
         $fld2(r) = fld2(v)$ 
    THEN
         $v := r$ 
    END

```

If we don't care about the value of some field (e.g., $fld2$), we simply omit any guard on that field as follows:


```

Ev1 =
    ANY  $r$  WHERE
         $r \in R$ 
         $fld1(r) = e1$ 
    THEN
         $v := r$ 
    END

```

Sometimes we will wish to model a set of structured elements as a machine variable, e.g.,

```

INVARIANT  $vs \subseteq R$ 

```

We can add a structured element to this set using the following event:

```

AddElement =
    ANY  $r$  WHERE
         $r \in R$ 
         $fld1(r) = e1$ 
         $fld2(r) = f1$ 
    THEN
         $vs := vs \cup \{r\}$ 
    END

```

12.1.2 Extending Structured Types

In a refinement we can introduce new fields to an existing structured type. Suppose R is defined in context $C1$ with fields $fld1$ and $fld2$ as before. For the refinement, suppose we wish to add a field $fld3$ of type $TP3$ to structured type R . Let us use the following invented syntax for this (not part of Event-B syntax):

```

EXTEND RECORD  $R$  WITH  $fl3 \in TP3$ 

```

This can be done by including the new field g as projection function on R in a context $C2$ (that extends $C1$). The new field is defined as follows:

SETS	$TP3$
CONSTANTS	$fld3$
AXIOMS	$fld3 \in R \rightarrow TP3$

Above, we saw the *AddElement* event that adds a structured element to the set *vs*. We may want to specify a value for the new *fld3* field in the refinement of this event. Using the event extension mechanism, this can be achieved as follows:

AddElement	=
EXTENDS	<i>AddElement</i>
WHEN	
	$fld3(r) = g1$
THEN	
	<i>skip</i>
END	

This is a form of superposition refinement where we add more data structure to the state by adding a new field to elements of the structured type R . In the above event we strengthen the specification by adding a guard using the event extension mechanism. The event extension mechanism means that this short form is the same as specifying the refined event as follows:

```

AddElement = REFINES AddElement
ANY  $r$  WHERE
     $r \in R$ 
     $fld1(r) = e1$ 
     $fld2(r) = f1$ 
     $fld3(r) = g1$ 
THEN
     $vs := vs \cup \{r\}$ 
END

```

12.1.3 Sub-Typing

Extension, as described above, is used when adding new fields to a structured type as part of a refinement step. That is, at different abstraction levels, we may have different numbers of fields in a structured type.

Sometimes we wish to have different subtypes of a structure at the same abstraction level. We can achieve this by defining subsets of the structure type and defining projection functions for those subsets only. For example suppose we define a message type that has sender and receiver fields as well as a message identifier. In our invented notation this would be:

```

RECORD    Message ::
     $sender \in Agent$  ,
     $receiver \in Agent$  ,
     $ident \in Ident$  ;

```

In Event-B this would be:

SETS*Message, Agent, Ident***CONSTANTS***sender, receiver, ident***AXIOMS***sender* $\in$ *Message* $\rightarrow$ *Agent**receiver* $\in$ *Message* $\rightarrow$ *Agent**ident* $\in$ *Message* $\rightarrow$ *Ident***END**

We wish to have two subtypes of message, request messages and confirmation messages. We define two subsets of Message as follows:

CONSTANTS*ReqMessage, ConfMessage***AXIOMS***ReqMessage* $\subseteq$ *Message**ConfMessage* $\subseteq$ *Message***END**

We require requests to have a product and an amount field and a confirmation to have a price field. In our invented notation this would be specified as follows:

EXTEND RECORD *ReqMessage* **WITH***product* $\in$ *Product**amount* $\in$ $\mathcal{N}$ **EXTEND RECORD** *ConfMessage* **WITH***price* $\in$ $\mathcal{N}$

In Event-B the additional fields are specified as projection functions on the relevant message subsets:

CONSTANTS <i>product, amount, price</i> AXIOMS <i>product</i> $\in$ <i>ReqMessage</i> $\rightarrow$ <i>Product</i> <i>amount</i> $\in$ <i>ReqMessage</i> $\rightarrow \mathcal{N}$ <i>price</i> $\in$ <i>ConfMessage</i> $\rightarrow \mathcal{N}$ END

Typically we would require that the message subtypes are disjoint. In that case we would add an additional axiom:

AXIOMS <i>ReqMessage</i> $\cap$ <i>ConfMessage</i> = $\emptyset$ END

If we know that there are no other subtypes besides requests and confirmations and that they should be disjoint, then we can condense the subset and disjointness axioms into a partitions axiom:

AXIOMS <i>partition</i>(<i>Message</i>, <i>ReqMessage</i>, <i>ConfMessage</i>) END

Suppose we have 2 variables representing request and confirmation channels respectively:

INVARIANT

$$req\_channel \subseteq ReqMessage$$

$$conf\_channel \subseteq ConfMessage$$

The following *Confirm* event models an agent responding to request by issuing a confirmation message. The *Confirm* event is enabled for the agent if there is a request message, *req*, whose receiver is that agent. The effect of the event is to chose a confirmation message, *conf*, whose fields are constrained with appropriate guards and add that confirmation message to the confirmations channel:

Confirm =**ANY** *req, conf, agent* **WHERE***req* ∈ *req_channel**agent* = *receiver(req)**conf* ∈ *ConfMessage**sender(conf)* = *agent**receiver(conf)* = *sender(req)**price(conf)* = *calculate_price(product(req) ↦ amount(req))***THEN***conf_channel* := *conf_channel* ∪ {*conf*}**END**

It might seem that we should always specify that subtypes are disjoint, but this is not so. Not making them disjoint allows us to represent a form of 'multiple inheritance'. For example, we could declare a new subtype *Req-ConfMessage* that subsets both *ReqMessage* and *ConfMessage*:

CONSTANTS*ReqConfMessage***AXIOMS** $ReqConfMessage \subseteq ReqMessage$ $ReqConfMessage \subseteq ConfMessage$ **END**

Elements of *ReqConfMessage* will have the fields of both subtypes, i.e., *sender*, *receiver*, *ident*, *product*, *amount* and *price*.

This combined subtype could be further extended:

EXTEND RECORD *ReqConfMessage* **WITH**
time $\in$ *TIME*

For an example of a Rodin development that uses the projection-based approach to modelling and extending records see [REB07].

12.1.4 Constructor-based Records

Constructor-based Records

An alternative style of modelling records is to define constructor functions that construct a record from a tuple of the components of that record. This is the approach that VDM follows, for example. Suppose we wish to model the following record structure:

RECORD *R* ::
fld1 $\in$ *TP1* ,
fld2 $\in$ *TP2* ;

We introduce the constructor function for *R* records, *mk* – *R*, as follows:

SETS $R, TP1, TP2$ **CONSTANTS** $mk-R, fld1, fld2$ **AXIOMS** $mk-R \in (TP1 \times TP2) \mapsto R$ $fld1 \in R \rightarrow TP1$ $fld2 \in R \rightarrow TP2$ $\forall x, y \cdot fld1(mk-R(x \mapsto y)) = x$ $\forall x, y \cdot fld2(mk-R(x \mapsto y)) = y$ **END**

The $mk-R$ function is specified as a constructor for type R that takes an element of type $TP1$ and an element of type $TP2$ and constructs a value of type R . The constructor is specified as injective because when the components are different, then the corresponding records should be different. The above declaration also makes the constructor surjective which means that the constructor provides a way of generating all records of type R .

With the constructor based approach, the *AddElement* event already shown above does not require a selection of a record satisfying the required properties. Instead we use the constructor directly:

```

AddElement  =
                BEGIN
                     $vs := vs \cup \{ mk-R(e1, f1) \}$ 
                END

```

It is possible to define different subtypes in a similar way as in the projection-based approach by defining separate constructors for each (dis-joint) subtype.

Adding new fields in a refinement is more difficult with the constructor approach however. We would need to introduce a completely new constructor for the full extended record structure and to get the refinement to work

correctly, we would need to define a relationship between the original record and the extended version. This merits further exploration.

12.1.5 Projections versus Constructors

As we have seen, the constructor approach can lead to more succinct event specifications than the projection approach. The downside of the constructor approach is that it is difficult to extend records in refinement steps and stepwise refinement and introduction of structure through refinement is an major feature of the Event-B approach.

We could say that the projection approach gives us *open record* structures, that is, structures that are easy to extend in a refinement. The constructor approach gives us *closed records* that are not directly extendable in refinement steps.

The constructor approach can be seen as a strengtening of the projection approach as is now explained. Consider again the two projection functions introduced to represent fields of the record structure R in the projection approach:

AXIOMS

$$fld1 \in R \rightarrow TP1$$

$$fld2 \in R \rightarrow TP2$$

END

The direct product of these two projections ($fld1 \otimes fld2$) has the following type:

$$fld1 \otimes fld2 \in R \rightarrow (TP1 \times TP2)$$

The inverse of the direct product $(fld1 \otimes fld2)^{-1}$ relates pairs of type $TP1 \times TP2$ to elements of type R . If we specify that $fld1 \otimes fld2$ is injective, i.e., its inverse is a function, then $(fld1 \otimes fld2)^{-1}$ becomes a constructor function for records of type R .

This means that an open record structure can be 'closed-off' by the addition of an axiom stating that the direct product of the fields is injective:

AXIOMS (*Closing*)
 $fld1 \otimes fld2 \in R \mapsto (TP1 \times TP2)$
END

Once this *closing axiom* is introduced, we can no longer introduce new fields to the record structure - at least not without introducing an inconsistency. If we were to now add a field *fld3*, the closing axiom would mean that records whose *fld1* and *fld2* fields are the same, must be correspond to the same record *r* even if they differ in the *fld3* field. But since *fld3* is a function, different *fld3* values cannot be related to the same record *r*.

Regardless of whether the closing axiom is introduced, some feasibility proof obligations arising from the use of the projection approach will require the existence of a record corresponding to any tuple of field components. This can be specified by the following *feasibility* axiom stating that the product of the projections is surjective:

AXIOMS (*Feasibility*)
 $fld1 \otimes fld2 \in R \twoheadrightarrow (TP1 \times TP2)$
END

12.1.6 Recursive Structured Types

It is well known how to deal with recursive records using the constructor approach. This is explored further in the context of Event-B and Rodin as part of the mathematical extension work where proposals for inductive datatypes are made (see Mathematical Extensions). Inductive datatypes are defined by constructor functions similar to constructor-based records. For example, to define lists we would use two disjoint constructors, *nil* and *cons*. The *cons* constructor for lists typically comes with two projections, *head* and *tail*. The structural induction principle for inductive datatypes is well known.

With the projection approach we can define two subtypes, for empty and non-empty lists, and extend the non-empty subtype with head and tail projections.

SETS $T, Node, LIST$ **CONSTANTS** $Nil, Cons$ **AXIOMS** $partition(Node, Nil, Cons)$ **END****EXTEND RECORD $Cons$ WITH** $head \in T$ $tail \in LIST$

The appropriate induction principle for the projection based approach needs to be explored further.

12.2 Term Rewriting in Logics of Partial Functions

In this chapter we summarize the contributions of the DEPLOY project to term rewriting in logics of partial functions. The main contribution is *directed rewriting*, a technique that avoids solving well-definedness conditions during term rewriting. Directed rewriting already appears in ML, an automated theorem prover for classical B, and PVS [ORS92], although we are not aware of a written documentation or justification. Butler and Maamria [MB10b] develop a theoretical foundation for a restricted version of directed rewriting. Schmalz [Sch11b] shows how to overcome these limitations and provides a positive report on its practical impact. Although the technique of directed rewriting is described in the context of Event-B, it has applications in other logics of partial functions such as LPF [JM94], the logic underlying VDM, and PVS. See [Sch11b] for detailed explanations.

In Section 12.2.1 we recall how Event-B treats partial functions. The problems related to term rewriting in the presence of partial functions are illustrated in Section 12.2.2. In Section 12.2.3 we describe directed rewriting, i.e., the technique used by Rodin to avoid these problems to a large extent.

12.2.1 Event-B – a Logic of Partial Functions

We introduce the concepts on an informal level that are necessary to understand the remaining chapter. A comprehensive description of Event-B's logic can be found in [Sch11b, Sch11a]. Event-B's logic shares many properties with higher-order logic (HOL, see [GM93]): it has notions of set and set comprehensions, Russel's paradox is avoided by a type discipline, and it admits quantification over sets. There are also a few differences, most notably the fact that functions in Event-B are partial by default and the application of a partial function to arguments outside of its domain results in an *ill-defined* term. The opposite of ill-defined is *well-defined*. While $1 / 0$ ("one divided by zero") is ill-defined in Event-B, the HOL counterpart $1 \text{ div } 0$ denotes a possibly unknown integer¹.

We decide to view an ill-defined term as a term denoting a special fault-value instead of a term that does not denote at all. In particular, all ill-defined terms of a given type have the same denotation. For convenience, we introduce a constant $\bullet$, called *ill-definedness*, that denotes this fault-value. The *well-definedness* operator D indicates whether a term is well-defined: $D(t)$ is true if t is well-defined and otherwise false. Note that $\bullet$ and D have been introduced to facilitate theoretical investigations, but are not supported by Rodin.

An operator f is *strict* iff well-definedness of $f(t_1, \dots, t_n)$ implies well-definedness of $t_1, \dots, t_n$. The opposite of strict is *lazy*. Most operators in Event-B are the *strict extensions* of their classical counterparts, i.e., they yield the usual result according to their classical counterparts if applied to well-defined arguments and ill-definedness otherwise. Examples of operators that are strict extensions include *equality* $=$, *intersection* $\cap$, and *membership* $\in$.

As an example of a partial operator consider the *integer division* operator $/$: the term t / u is well-defined iff $D(t) \wedge D(u) \wedge u \neq 0$. If t / u is well-defined, its denotation is obtained by dividing t by u in rational arithmetic and rounding towards zero. Note that division is strict.

The boolean connectives available in Event-B are $\top$ (*truth*), $\perp$ (*falsity*), $\wedge$ (*conjunction*), $\vee$ (*disjunction*), $\Rightarrow$ (*implication*), and $\Leftrightarrow$ (*equivalence*). Truth and falsity have their usual denotations. Negation and equivalence are the strict extensions of their classical counterparts. The denotation of conjunc-

¹In Isabelle/HOL [NPW10], $1 \text{ div } 0$ equals 0.

tion is chosen according to the following truth table:

$\wedge$	$\top$	$\perp$	$\bullet$
$\top$	$\top$	$\perp$	$\bullet$
$\perp$	$\perp$	$\perp$	$\perp$
$\bullet$	$\bullet$	$\perp$	$\bullet$

Disjunction and implication are defined such that $\varphi \vee \psi$ has the same denotation as $\neg(\neg\varphi \wedge \neg\psi)$ and $\varphi \Rightarrow \psi$ the same denotation as $\neg\varphi \vee \psi$.

Event-B's proof calculus is organized in terms of *sequents* of the form $\psi_1, \dots, \psi_n \vdash \varphi$, where the formulae ψ_i are called *hypothesis* and the formula φ is called *goal*. Clearly, if a hypothesis is false or the goal is true, the sequent is true. Also, if all hypotheses are true and the goal is false, then the sequent is false. Finally, if a hypothesis or the goal is ill-defined, the sequent is true. This last choice may appear arbitrary. It is unclear whether a different sequent semantics would have advantages over the current one; but the current semantics is in fact essential for soundness of directed rewriting.

12.2.2 Rewriting Terms to Equivalent Terms

Although the formulae $t = t$ and $\top$ have the same denotations in classical logics such as classical first- or higher-order logic or Zermelo-Fraenkel set theory, the denotations may differ under Event-B's partial function semantics – take $\bullet$ for t . The soundness of term rewriting is typically justified by the observation that terms are replaced only by terms with the same denotation. If we stick with this justification of term rewriting, we may not rewrite $t = t$ to $\top$ in general, but instead only after verifying $D(t)$.

The phenomenon that rewrite rules from classical logics become unsound when transferred naively to Event-B's logic is quite common, even for rules that do not involve partial functions. We denote a *symmetric rewrite rule* by $t \equiv u$ and say that it is *sound* iff t and u have the same denotation. Then, the following examples of symmetric rewrite rules are all unsound:

$$t = t \equiv \top, \quad (12.1)$$

$$t \in R \cap S \equiv t \in R \wedge t \in S, \quad (12.2)$$

$$t \in \emptyset \equiv \perp. \quad (12.3)$$

The unsoundness stems from the special status of $\bullet$, i.e., (12.1) is unsound because equality is strict and therefore not reflexive, (12.2) is unsound because intersection is strict and conjunction is lazy, and (12.3) is unsound because membership is strict.

In some cases, unsoundness could be avoided by choosing appropriate semantics, i.e., making equality reflexive, intersection lazy, and so on. However, this would result in difficulties when applying Event-B to the problems it has initially been designed for: for example, if we use ill-definedness to model exceptions, then a reflexive equality would not correspond to the equality in Java.

Soundness can also be recovered by imposing appropriate well-definedness preconditions, e.g., $D(t)$ in the case of (12.1) and $D(R) \wedge D(S)$ in the case of (12.2). It is however challenging to setup a tactic solving these preconditions in an efficient and reliable manner.

12.2.3 Directed Rewriting

The main idea of *directed rewriting* is to abandon the standard justification for soundness. Instead of only rewriting terms to terms with the same denotation, we also rewrite ill-defined to well-defined terms. Formally, a *directed rewrite rule* is denoted $t \sqsubseteq u$, where t and u are terms of the same type, and $t \sqsubseteq u$ is *sound* iff t is ill-defined or t and u have the same denotations. The relation $\sqsubseteq$ constitutes a partial order, which is also known as *flat domain order* (see e.g. [Pau87, p. 61]).

Soundness of directed rewriting is a consequence of the fact that the operators and binders available in Event-B are monotonic w.r.t. $\sqsubseteq$ and the semantics of sequents has been chosen such that sequents with an ill-defined hypothesis or goal are considered valid. The details of the soundness proof, including for *conditional* directed rewriting, can be found in [Sch11b].

The major limitation of directed rewriting is that only the arguments of monotonic operators and binders can be rewritten. A common example of a non-monotonic operator is the well-definedness operator D : although $\bullet \sqsubseteq \top$ is sound, it would be unsound to rewrite $D(\bullet)$ to $D(\top)$. We admit that the well-definedness operator is useful for specifying properties of meta-logical nature (see 12.2.6), but it seems to be dispensable when specifying software systems. We therefore view the monotonicity restriction as acceptable.

12.2.4 Practical Relevance

The reader may want to verify that the (unsound) rules in (12.1–12.3) can all be restated as sound directed rewrite rules. Additionally, we have analyzed the rewrite rules available in the Rodin platform. The details of the analysis can be found in [Sch11c]. In total, Rodin implements 453 (unconditional) directed rewrite rules; 165 of them (36%) would be unsound if stated as symmetric rewrite rules. In other words, if directed rewriting was unknown

to us, 36% of Rodin’s rewrite rules would need to be made conditional to ensure soundness. Thus, directed rewriting makes conditional rewrite rules unconditional in a significant number of cases and constitutes an important optimization of Rodin’s term rewriter.

The reader may have the impression that directed rewriting mainly compensates for problems introduced by the fact that Event-B’s logic explicitly supports partial functions. But this is not entirely true. In logics of total functions it is quite common to approximate partial functions by underspecified total functions. In such a logic, $x / 0$ denotes an unspecified integer. Therefore x / x equals 1 only if $\neg (x = 0)$. In Event-B and with directed rewriting, we can avoid the condition $\neg (x = 0)$ by restating the rule as $\$x / \$x \sqsubseteq 1$. Thus, directed rewriting not only compensates for problems introduced by explicit partiality, but also makes rules unconditional that are commonly conditional in logics of total functions. In the case of Rodin, there are 35 such rules.

12.2.5 Safety

For automated theorem proving we are interested in *safe* rules, i.e., rules that never transform a valid sequent into an invalid one and can thus be applied blindly. Recall that a sequent is valid if some hypothesis or the goal is ill-defined. Thus, directed rewriting is unsafe in general: As an example, consider the sound directed rewrite rule $\bullet \sqsubseteq \perp$, which transforms the valid sequent $\vdash \bullet$ into the invalid sequent $\vdash \perp$.

Although directed rewriting is unsafe, the examples witnessing this unsafety seem artificial and never occurred to us in practice. We have investigated this phenomenon and discovered that the safe inference rules implemented by Rodin maintain an invariant that entails safety of directed rewriting [Sch11b]. In a nutshell, we show that directed rewriting in Rodin can be made safe after some mild modifications of the proof obligation generator and provided that only safe inference rules are used during the proof attempt. We also point out how safety of directed rewriting can be maintained when implementing new inference rules.

12.2.6 Proving User Supplied Rules Sound

Rodin provides a generic term rewriter to which the user can supply new rewrite rules [MB10b]. The term rewriter accepts a new rule only if the user formally proves its soundness. Rules with lazy operators, such as conjunction, disjunction, and implication, on the left-hand side are however rejected,

because it has been unclear how to generate the required soundness proof obligations. Support for rules involving binders is limited, too.

It is natural to specify the soundness proof obligation of a rewrite rule using the well-definedness operator D and *operator variables*, i.e., variables that match possibly ill-defined terms: the rule $t \sqsubseteq u$ is sound iff $\vdash t = u$ and $D(t) \vdash D(u)$ are valid. It would however be difficult to add the well-definedness operator D to the logic implemented by Rodin, because D is not monotonic and monotonicity assumptions are hard-wired in several places, in particular in Rodin's term rewriter. We have therefore developed an embedding of the fragment of Event-B with operator variables and D into the fragment of Event-B without [Sch11b]. The main challenge was to cope with terms that contain both D and operator variables.

12.3 Theory Plug-in and Mathematical Extensions

This chapter provides an overview of the work carried out to enhance the extensibility of Event-B toolset. In particular, we outline the approach that is adopted to enable the users of Rodin to contribute to the Event-B mathematical language and the proof infrastructure.

12.3.1 Objectives

Our aim is to improve the overall extensibility of Event-B to enhance usability and effectiveness of the methodology. We are primarily concerned with facilitating the addition of new operators (i.e., *language extensions*) and new proof rules (i.e., *prover extensions*) to suit end-users needs. It is essential to ensure that any technique that achieves the aforementioned targets has to maintain *practicality of use* and ensure *soundness preservation*. Practicality of use is important to relieve end-users from writing Java code. Soundness preservation ensures that any extensions do not compromise the logical foundations of the formalism. The following key points summarise the objectives of this work:

1. provide a mechanism by which users can define polymorphic operators in a familiar fashion thereby allowing language extensions. The new mechanism needs to adhere to the aforementioned requirements: practicality of use and soundness preservation.
2. provide a mechanism by which the Rodin proving infrastructure can be augmented with new proof rules. Any newly added rules will have to

be validated so that the soundness of the existing prover is not compromised. Rewrite and inference rules are used in Rodin to discharge proof obligations. The following milestones are important in order to achieve this objective:

- (a) provide a unifying study of term rewriting and well-definedness. This is of major importance since the Event-B logic deals with partial functions which may give rise to potentially ill-defined term.
- (b) study how rewriting can be integrated as a proof step within the well-definedness preserving sequent calculus. We study how conditional rewrite rules can be used alongside the well-definedness preserving inference rules in order to enhance the proving capabilities of Rodin.

12.3.2 The Theory Construct

We recall that modelling in Event-B is carried out by means of contexts and machines. Theories [MBER10] are Event-B constructs which are similar in their morphology to contexts and machines. A theory acts as a place-holder for mathematical and prover extensions. The following listing describes the overall structure of Event-B theories.

```
theory  name
type parameters   $T_1, \dots, T_n$ 

{   $\langle \textit{Datatype Definition} \rangle$ 
  |  $\langle \textit{Operator Definition} \rangle$ 
  |  $\langle \textit{Polymorphic Theorem} \rangle$ 
  |  $\langle \textit{Metavariables} \rangle$ 
  |  $\langle \textit{Rewrite Rule} \rangle$ 
  |  $\langle \textit{Inference Rule} \rangle$  }
```

An Event-B theory has a name which identifies it within the workspace. A theory can have an arbitrary number of type parameters which are pairwise distinct, in which case the theory is said to be *polymorphic* on its type parameters. A theory may also contain an arbitrary number of definitions and rules. In this chapter, we focus on new operator definitions. We will also describe proof rules and polymorphic theorems and how they can be specified and validated through the theory construct.

Soundness Preservation

In the process of defining new extensions (e.g., new operator or a new rewrite rule), it is possible to introduce unsoundness to the prover. As such, it is imperative that the ease of use of the theory component is complemented by an effective measure to discover and eliminate any soundness-threatening extensions. Furthermore, we argue that such measure should not hinder the usability of any provided tool support.

The use of proof obligations is widespread in many formal techniques not least in Event-B. In the case of Event-B modelling, proof obligations provide simple semantics by which it is possible to understand the system being modelled [Hal08]. We argue that using proof obligations to verify any user-defined extensions will ensure that potentially unsound extensions are brought to the attention of the user. Moreover, since modellers are *familiar* with the use of proof obligations in contexts and machines, this approach achieves a good balance between effectiveness and usability. Throughout the rest of this chapter and subsequent chapters, whenever a new extension is introduced, any required proof obligations are singled out and their adequacy is justified.

Theory Deployment

We distinguish between two separate but intrinsically linked activities in the context of Event-B theories. *Theory development* refers to the activity of defining and validating theories. At this stage, extensions are defined and proof obligations are generated for each extension as required. This activity may follow an iterative pattern since inspecting failed automatic proof attempts may reveal important information about the soundness or otherwise of extensions. This will guide the modeller to change definitions accordingly. Therefore, theory development greatly benefits from the reactive nature of the Rodin platform[Meh07, ABHV06].

Theory deployment refers to the activity of making developed theories available for use in modelling. Theory deployment ensures that proof obligations are at least inspected by the user, and once deployed, any mathematical extensions and proof rules can be used to specify Event-B contexts and machines. As an example, consider a theory of boolean operators. The user may specify the usual operators (e.g., logical AND), define some inference and rewrite rules, and attempt to discharge any generated proof obligations. Once the user is satisfied with his/her own theory in terms of soundness, the theory can be deployed and used within a model that specifies an electric circuit. The use of theory-defined proof rules and polymorphic theorems

enables the user to reason at the level of mathematical extensions without detour through the classical (wired) Event-B mathematical language.

12.3.3 Event-B Mathematical Language

In the Event-B mathematical language [MV09] (Event-B inner syntax), terms (expressions) and formulae (predicates) are separate syntactic categories. Terms are defined using constants (e.g., 1), variables and operators (e.g., $\cup$). Term operators can have terms as arguments. They can also have formulae as arguments e.g., $(\lambda x \cdot P(x) \mid E(x))$ where $P(x)$ is a formula and $E(x)$ is a term.

Formulae, on the other hand, are built from basic formulae e.g., $x \in S$, logical connectives and quantifiers. Basic formulae take terms as arguments e.g., $x \in S$ has x and S as arguments.

Terms have a type which can be one of the following:

1. a basic set such as $\mathbb{Z}$ or a carrier set supplied by the modeller in contexts;
2. a power set of another type;
3. a cartesian product of two types.

Term operators have typing rules of the form:

$$\frac{\mathbf{type}(x_1) = \alpha_1 \dots \mathbf{type}(x_n) = \alpha_n}{\mathbf{type}(op(x_1, \dots, x_n)) = \alpha}.$$

Arguments of a basic formula must satisfy its typing rule e.g., the typing rule for the basic formula $finite(R)$ is:

$$\mathbf{type}(R) = \mathcal{P}(\alpha).$$

Alongside typing rules, term operators have well-definedness formulae. $\mathbf{WD}(E)$ is used to denote the well-definedness formula of term E . Proof obligations are generated (if necessary) to establish the well-definedness of terms appearing in models. To illustrate, we consider the term $card(E)$ for which we have:

$$\mathbf{WD}(card(E)) \Leftrightarrow \mathbf{WD}(E) \wedge finite(E).$$

Note. For the rest of this thesis, we use the term ‘*mathematical language*’ to refer to Event-B inner syntax that is wired. The term ‘*existing mathematical language*’ refers to the mathematical language augmented with any previously defined operator extensions.

12.3.4 Operator Definition

A new Event-B polymorphic operator can be defined by providing the following information:

1. *Parser Information*: this includes the syntax, the notation (infix or prefix), and the syntactic class (term or formula).
2. *Type Checker Information*: this includes the types of the child arguments, and the resultant type if the operator is a term operator.
3. *Prover Information*: this includes the well-definedness of the operator as well as its definition which may be used to reason about it.

The following snippet describes the general structure of a new operator definition:

```
operator syntax
  (prefix | infix)
args  $x_1 \in T_{x_1}, \dots, x_n \in T_{x_n}$ 
condition  $P(x_1, \dots, x_n)$ 
definition  $Q(x_1, \dots, x_n)$ 
```

1. ‘*syntax*’: defines the syntax of the new operator. It has to be distinct from previously used operator syntaxes as our approach does not allow operator overloading.
2. ‘**prefix**’ or ‘**infix**’: define the type of the notation that will be used for this operator either infix (e.g., $a \text{ op } b$) or prefix (e.g., $\text{op}(a, b)$).
3. ‘**args**’: defines the arguments of the operator. Each argument must have a name and a type. Names of the arguments are pairwise distinct.
4. ‘**condition**’: provides the well-definedness condition to be generated for this operator. We will show later how concrete well-definedness conditions are correctly generated from the above definition.
5. ‘**definition**’: provides the direct definition of the operator in terms of the existing mathematical language. The syntactic class of the operator is inferred from the syntactic class of $Q(x_1, \dots, x_n)$. If $Q(x_1, \dots, x_n)$ is a term, then the resultant type of the operator is type of $Q(x_1, \dots, x_n)$.

Example: The Sequence Operator

A sequence is an ordered list of objects where the same object can occur multiple times at different positions. It is, therefore, easy to see that a sequence can be defined as a polymorphic operator. The following snippet provides a definition of a sequence in Event-B.

```
theory SeqThy
type parameters S
operator Seq
  (prefix)
  args  $a \in \mathcal{P}(S)$ 
  condition  $\top$ 
  definition  $\{f, n \cdot f \in 1..n \rightarrow a \mid f\}$ 
```

In the above snippet, $1..n$ denotes a contiguous integer range. The previous definition describes the set of all sequences of the set a ; each sequence is defined as a total function from an integer range to the set a . The following typing rule is generated for the operator *Seq*:

$$\frac{\mathbf{type}(a) = \mathcal{P}(S)}{\mathbf{type}(\mathit{Seq}(a)) = \mathcal{P}(\mathcal{P}(\mathbb{Z} \times S))}.$$

In the following snippet, the formula operator *EmptySeq* takes a sequence, and ‘returns’ whether the sequence is empty. The term operators *HeadSeq* and *TailSeq* calculate the head and the tail of a non-empty sequence respectively.

```
operator EmptySeq
  (prefix)
  args  $s \in \mathcal{P}(\mathbb{Z} \times S)$ 
  condition  $s \in \mathit{Seq}(S)$ 
  definition  $\mathit{card}(s) = 0$ 

operator HeadSeq
  (prefix)
  args  $s \in \mathcal{P}(\mathbb{Z} \times S)$ 
  condition  $\neg \mathit{EmptySeq}(s)$ 
  definition  $s(1)$ 

operator TailSeq
  (prefix)
```

args $s \in \mathcal{P}(\mathbb{Z} \times S)$
condition $\neg \text{EmptySeq}(s)$
definition $\{i \cdot i \in 1..(\text{card}(s) - 1) \mid i \mapsto s(i + 1)\}$

The following typing rule is generated for the *EmptySeq* formula operator:

$$\mathbf{type}(s) = \mathcal{P}(\mathbb{Z} \times S) .$$

The following typing rules are generated for the head and tail operators:

$$\frac{\mathbf{type}(s) = \mathcal{P}(\mathbb{Z} \times S)}{\mathbf{type}(\text{HeadSeq}(s)) = S} ,$$

$$\frac{\mathbf{type}(s) = \mathcal{P}(\mathbb{Z} \times S)}{\mathbf{type}(\text{TailSeq}(s)) = \mathcal{P}(\mathbb{Z} \times S)} .$$

12.3.5 Operator Properties

In this subsection, we describe the different aspects of a new operator definition. More specifically, we focus on well-definedness, associativity and commutativity.

Well-Definedness

An important aspect of defining an operator is the well-definedness condition to be used. A simple strategy may use the well-definedness of the operator's direct definition. An advantage of a user-supplied condition is the possibility of strengthening well-definedness conditions to simplify proofs. In order to ensure that a supplied condition is in fact stronger than the default (i.e., the one inferred from the direct definition), a proof obligations is generated.

We recall the structure of an operator definition:

operator *syntax*
 (prefix | infix)
args $x_1 \in T_{x_1}, \dots, x_n \in T_{x_n}$
condition $P(x_1, \dots, x_n)$
definition $Q(x_1, \dots, x_n)$

An important property of well-definedness conditions is that they are themselves well-defined, i.e.,:

$$\mathcal{D}(\mathcal{D}(P)) \Leftrightarrow \top \quad \text{for any formula or term } P.$$

There is a possibility that the supplied well-definedness condition may not, in some cases, be well-defined (e.g., *HeadSeq* well-definedness condition). Therefore, the complete well-definedness condition of an operator is the following:

$$\mathcal{D}(P(x_1, \dots, x_n)) \wedge P(x_1, \dots, x_n) .$$

As an example, the well-definedness condition of the *HeadSeq* (and, coincidentally, *TailSeq*) operator is

$$s \in Seq(S) \wedge \neg EmptySeq(s) .$$

To ensure that the supplied well-definedness condition is stronger than the default one, the following proof obligations is generated:

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T_{x_1}, \dots, x_n \in T_{x_n} \cdot (\mathcal{D}(P(x_1, \dots, x_n)) \wedge P(x_1, \dots, x_n)) \Rightarrow \mathcal{D}(Q(x_1, \dots, x_n))}$$

As an example, the following proof obligations are generated for *EmptySeq* and *HeadSeq*, respectively:

$$\vdash_{\mathcal{D}} \forall s \in \mathcal{P}(\mathbb{Z} \times S) \cdot s \in Seq(S) \Rightarrow finite(s)$$

$$\vdash_{\mathcal{D}} \forall s \in \mathcal{P}(\mathbb{Z} \times S) \cdot (s \in Seq(S) \wedge \neg EmptySeq(s)) \Rightarrow (s \in \mathbb{Z} \text{ functional } S \wedge 1 \in dom(s))$$

using the following expansions:

$$\begin{aligned} \mathcal{D}(card(s)) &\hat{=} finite(s) , \\ \mathcal{D}(s(1)) &\hat{=} s \in \mathbb{Z} \text{ functional } S \wedge 1 \in dom(s) . \end{aligned}$$

Commutativity

An operator is said to be commutative if it can have *at least* two arguments of the same type, and the following formula is valid:

$$\begin{aligned} Q(x_1, x_2) &= Q(x_2, x_1) && \text{if the operator is a term operator, or} \\ Q(x_1, x_2) &\Leftrightarrow Q(x_2, x_1) && \text{if the operator is a formula operator.} \end{aligned}$$

Example. Consider the definition of the multiplication operator with a stronger well-definedness condition:

operator *mult* commutative
(infix)
args $x_1 \in \mathbb{Z}, x_2 \in \mathbb{Z}$
condition $x_1 \neq 0 \wedge x_2 \neq 0$
definition $x_1 * x_2$

In this case, the following formula describes the condition which asserts that the *mult* operator is commutative:

$$\forall x_1 \in \mathbb{Z}, x_2 \in \mathbb{Z} \cdot x_1 * x_2 = x_2 * x_1 .$$

More generally, if an operator is tagged by the user as commutative, then:

1. if the operator is a term operator, the following proof obligation is generated:

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T, x_2 \in T \cdot Q(x_1, x_2) = Q(x_2, x_1)}$$

2. if the operator is a formula operator, the following proof obligation is generated:

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T, x_2 \in T \cdot Q(x_1, x_2) \Leftrightarrow Q(x_2, x_1)}$$

Associativity

A term operator is said to be associative if:

- it can have *at least* two arguments of the same type,
- the resultant type of the operator is the same as that of the arguments,
- the following formula is valid:

$$Q(Q(x_1, x_2), z) = Q(x_1, Q(x_2, z)) .$$

If a term operator is tagged as associative, the following proof obligations is generated:

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T, x_2 \in T, z \in T \cdot Q(Q(x_1, x_2), z) = Q(x_1, Q(x_2, z))}$$

In this chapter, we present our approach in dealing with issues related to prover extensibility in Event-B. As mentioned in §12.3.1, an important requirement of such approach is the practicality of use. More importantly, a mechanism must be in place to avoid compromising the soundness of the formalism. Furthermore, adding support for extending the Event-B mathematical language poses further challenges in terms of reasoning about new operators. In this chapter, we describe a practical approach to prover extensibility through the theory component described in §12.3.2.

12.3.6 Rewriting

The use of equations is central to mathematics. Rewriting provides a powerful mechanism for ‘dealing computationally with equations’ [DP01]. In this subsection, we show how rewrite rules are defined in the theory construct. We also present the different proof obligations that ensure soundness of defined rules. The theoretical results in [MB10c] provide the justification for the proof obligations related to rewrite rule definitions. At the end of this subsection, we provide some examples of rewrite rules.

Defining Rewrite Rules

In the Event-B mathematical language [Abr10, MV09], formulae and terms are distinguished as two separate syntactic categories. Furthermore, each term must have a type. In other literature, terms are referred to as expressions, and formulae are referred to as predicates. The following definition describes the syntactic properties of Event-B rewrite rules. Note the introduction of the typing constraint.

Definition 3 (Event-B Rewrite Rule) *An Event-B rewrite rule is of the form*

$$\begin{array}{c} lhs \rightarrow C_1 : rhs_i \\ \dots \\ C_n : rhs_n \end{array}$$

where:

1. $n \geq 1$,
2. lhs is not a variable,
3. lhs and rhs_i (for all i such that $1 \leq i \leq n$) are of the same syntactic class,
4. C_i (for all i such that $1 \leq i \leq n$) are formulae,
5. C_i and rhs_i (for all i such that $1 \leq i \leq n$) only contain free variables from lhs ,
6. lhs and rhs_i (for all i such that $1 \leq i \leq n$) have the same type if lhs is a term.

In the special case where $n = 1$ and C_1 is syntactically equal to $\top$, the rewrite rule is called *unconditional*. An Event-B rewrite rule is said to be *conditional* if it is not unconditional.

A definition of a rewrite rule is completed by specifying whether the rule should be applied automatically or interactively. In §12.3.6, we describe certain cases where an automatic application should not be allowed.

Validating Rewrite Rules

In the previous subsection, we defined the syntax of Event-B rewrite rules. In this sub-subsection, we describe the different proof obligations that ensure soundness of defined rewrite rules. Firstly, however, the following definition presents an important property of rewrite rules.

Definition 4 (Case-complete Event-B Rewrite Rule) *A rewrite rule*

$$\begin{aligned} lhs \rightarrow \quad & C_1 : rhs_1 \\ & \dots \\ & C_n : rhs_n \end{aligned}$$

is said to be case-complete if the following sequent is valid:

$$H \vdash_{\mathcal{D}} \bigvee_{i=1}^n C_i ,$$

where H is a formula providing typing information for all free variables of lhs .

Definition 4 ensures that rules are applied in a sound fashion as described in the previous chapter. Note that case-completeness is not a syntactic property, but one that requires mathematical proof. The following definition defines soundness in the context of Event-B rewrite rules.

Definition 5 (Sound Event-B Rewrite Rule) *An Event-B rewrite rule*

$$\begin{aligned} lhs \rightarrow \quad & C_1 : rhs_i \\ & \dots \\ & C_n : rhs_n \end{aligned}$$

is said to be sound if the following sequents are provable:

1. $H, \mathcal{D}(lhs), C_i \vdash_{\mathcal{D}} \mathcal{D}(rhs_i)$ for all i such that $1 \leq i \leq n$,

2. (a) $H, C_i \vdash_{\mathcal{D}} lhs = rhs_i$ for all i such that $1 \leq i \leq n$ if lhs is a term, or;
 (b) $H, C_i \vdash_{\mathcal{D}} lhs \Leftrightarrow rhs_i$ for all i such that $1 \leq i \leq n$ if lhs is a formula,
3. (a) $H \vdash_{\mathcal{D}} \bigvee_{i=1}^n C_i$ if the rule is case-complete,
 (b) $H, \mathcal{D}(lhs) \vdash_{\mathcal{D}} \mathcal{D}(C_i)$ for all i such that $1 \leq i \leq n$ if the rule is case-complete.

where H is a formula providing typing information for all free variables occurring in lhs .

Examples

Example 1. Assuming two variables x and y of the same type $\mathbb{Z}$, then the following is a rewrite rule

$$\begin{array}{l} (x - 1)(y - 1) \rightarrow x = 1 : 0 \\ y = 1 : 0 \end{array}$$

which is sound but not case-complete. Since the rule has more than one right hand side, it is conditional.

Example 2. In §12.3.4, we defined the sequence operator. To be able to reason about formulae containing the *Seq* operator, the following unconditional rewrite rule is generated:

$$Seq(a) \rightarrow \top : \{f, n \cdot f \in 1..n \rightarrow a \mid f\}$$

assuming a type parameter S and a variable a of type $\mathcal{P}(S)$.

12.3.7 Polymorphic Theorems

In an Event-B development, a modeller can specify certain properties of the system in question, and theorems can be used to ensure that the properties do capture the intended understanding of the system. Similarly, when defining a new operator as described in §12.3.4, the user might want to verify that the definition of the operator captures the intended meaning. We propose polymorphic theorems to achieve this effect. Moreover, we describe a mechanism by which these theorems can be incorporated in proofs which arise from models. We conclude this subsection by providing concrete examples.

Defining Polymorphic Theorems

The following definition describes the syntactic properties satisfied by polymorphic theorems.

Definition 6 (Event-B Polymorphic Theorem) *Let $\alpha_1, \dots$, and α_n be type parameters. A formula $P(\alpha_1, \dots, \alpha_n)$ is an Event-B polymorphic theorem if*

$$\text{Var}(P(\alpha_1, \dots, \alpha_n)) = \{\alpha_1, \dots, \alpha_n\} .$$

In this case, we say that the theorem $P(\alpha_1, \dots, \alpha_n)$ is polymorphic on each of the type parameters $\alpha_1, \dots$, and α_n .

In other words, a formula is a polymorphic theorem if its free variables are all type parameters.

Validating Polymorphic Theorems

Definition 6 describe the syntax of polymorphic theorems. The following definition presents the notion of soundness in the context of polymorphic theorems.

Definition 7 (Sound Event-B Polymorphic Theorem) *An Event-B polymorphic theorem $P(\alpha_1, \dots, \alpha_n)$ is said to be sound if the following sequents are provable:*

1. $\vdash_{\mathcal{D}} \mathcal{D}(P(\alpha_1, \dots, \alpha_n))$,
2. $\vdash_{\mathcal{D}} P(\alpha_1, \dots, \alpha_n)$.

Definition 7 ensures that polymorphic theorems are well-defined and valid. Note the similarity between the sequents in Definition 7 and the proof obligations related to theorems in Event-B contexts. The next subsection provides a justification for the previous definition.

Using Polymorphic Theorems

The *cut* rule

$$\frac{H \vdash_{\mathcal{D}} \mathcal{D}(P) \quad H \vdash_{\mathcal{D}} P \quad H, P \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} Q} \text{ cut}_{\mathcal{D}}$$

can be extremely useful when conducting proofs as it imitates the general approach taken when doing proofs in mathematics (i.e., using intermediate lemmas to guide proofs). In what follows, we show how the cut rule can provide a sound platform for using polymorphic theorems in Event-B proofs.

A formula P' is said to be an *instance* of the polymorphic theorem $P(\alpha_1, \dots, \alpha_n)$ if there exists a type substitution σ_t such that:

$$P' \cong \sigma_t(P(\alpha_1, \dots, \alpha_n)) . \quad (12.4)$$

where σ_t provides a substitution for all type parameters occurring in $P(\alpha_1, \dots, \alpha_n)$. An instance of a polymorphic theorem can be added as a hypothesis in a sequent as follows:

$$\frac{\boxed{H \vdash_{\mathcal{D}} \mathcal{D}(\sigma_t(P(\alpha_1, \dots, \alpha_n)))} \quad \boxed{H \vdash_{\mathcal{D}} \sigma_t(P(\alpha_1, \dots, \alpha_n))} \quad H, \sigma_t(P(\alpha_1, \dots, \alpha_n)) \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} Q} \text{cut}_{\mathcal{D}} .$$

If the polymorphic theorem $P(\alpha_1, \dots, \alpha_n)$ is sound as per Definition 7, then the boxed sequents can be removed and the polymorphic theorem can be used in proofs as follows:

$$\frac{H, \sigma_t(P(\alpha_1, \dots, \alpha_n)) \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} Q} \text{thm}_{\mathcal{D}} .$$

Examples

Example 1. The following formula is a sound Event-B polymorphic theorem:

$$\forall x, y \cdot x * y = 0 \Rightarrow (x = 0 \vee y = 0) .$$

Example 2. The following formulae are sound Event-B polymorphic theorems:

$$\begin{aligned} & \forall a \in \mathcal{P}(S) \cdot \emptyset \in \text{Seq}(a) , \\ & \forall a \in \mathcal{P}(S) \cdot (\forall s \in \mathcal{P}(\mathbb{Z} \times S) \cdot s \in \text{Seq}(a) \Rightarrow \text{finite}(s)) , \\ & \forall a \in \mathcal{P}(S) \cdot (\forall s \in \mathcal{P}(\mathbb{Z} \times S) \cdot (s \in \text{Seq}(a) \wedge \neg \text{EmptySeq}(s)) \Rightarrow \text{TailSeq}(s) \in \text{Seq}(a)) , \end{aligned}$$

assuming a type parameter S .

12.3.8 Inference Rules

In this subsection, we show how a special subset of polymorphic theorems can be manipulated in such a way that they can be used as inference rules. As mentioned earlier, polymorphic theorems achieve a two-fold objective. They can be used to ensure operator definitions capture the intended semantics. They can also be used in proofs as demonstrated in §12.3.7. We show that polymorphic theorem with a specific structure can be used in a similar way to inference rules.

Defining Inference Rules

The following definition describes the syntactic properties satisfied by inference rules.

Definition 8 (Event-B Inference Rule) *An Event-B inference rule is a pair $(\vec{G}, I)$ where:*

1. *I is an Event-B formula that is syntactically distinct from $\top$, called the infer clause,*
2. *$\vec{G}$ is a set of Event-B formulae, called the given clauses,*
3. *one of the following syntactic condition holds:*

$$\begin{aligned} \mathcal{V}ar(I) &\subseteq \bigcup_{H \in \vec{G}} \mathcal{V}ar(H) , \\ \bigcup_{H \in \vec{G}} \mathcal{V}ar(H) &\subseteq \mathcal{V}ar(I) . \end{aligned}$$

The previous definition describes the syntactic properties of inference rules. The infer clause has to be different from $\top$ since otherwise the inference cannot be of any use. Moreover, the third condition ensures that inference rules are at least applicable in one direction.

Definition 9 (Backward-applicable Event-B Inference Rule) *An Event-B inference rule $(\vec{G}, I)$ is said to be backward-applicable if the following condition holds:*

$$\bigcup_{H \in \vec{G}} \mathcal{V}ar(H) \subseteq \mathcal{V}ar(I) .$$

Definition 10 (Forward-applicable Event-B Inference Rule) *An Event-B inference rule $(\vec{G}, I)$ is said to be forward-applicable if the following condition holds:*

$$\mathcal{V}ar(I) \subseteq \bigcup_{H \in \vec{G}} \mathcal{V}ar(H) .$$

The previous two definitions outline the sufficient conditions under which it is possible to apply inference rules in one direction or the other.

Validating Inference Rules

As mentioned in the previous subsection, inference rules as defined above are a special case of implicative polymorphic theorems. For each inference rule, we can derive the appropriate polymorphic theorem.

Definition 11 (Derived Theorem) *The following formula is called the derived theorem of the Event-B inference rule $(\vec{G}, I)$:*

$$\forall \vec{x} . \vec{G} \Rightarrow I ,$$

where $\vec{x}$ are the free variables in all of $\vec{G}$ and I .

The following definition describes the sufficient conditions under which an inference rule is considered valid.

Definition 12 (Valid Inference Rule) *An inference rule $(\vec{G}, I)$ is said to be valid if its derived polymorphic theorem is sound.*

We recall that a polymorphic theorem is sound if it is provable and well-defined.

Using Inference Rules

Inference rules can be used in a backward style as well forward style. If used in backward style, it discharges or splits the goal. If applied in a forward style, more hypotheses get generated.

12.3.9 Tool Support

The Theory plug-in (currently on version 1.2 as of August 29, 2011) implements the different concepts outlined in this chapter. In brief, it enables the following:

- The specification of new polymorphic operators to enhance the expressiveness of Event-B.
- The specification of new datatypes. The theoretical foundations of datatypes are still being developed for Event-B. Nevertheless, the Theory plug-in provides good support for datatypes and recursive definitions.
- The specification of new proof rules. The Theory plug-in provides facilities to define and user rewrite and inference rules.

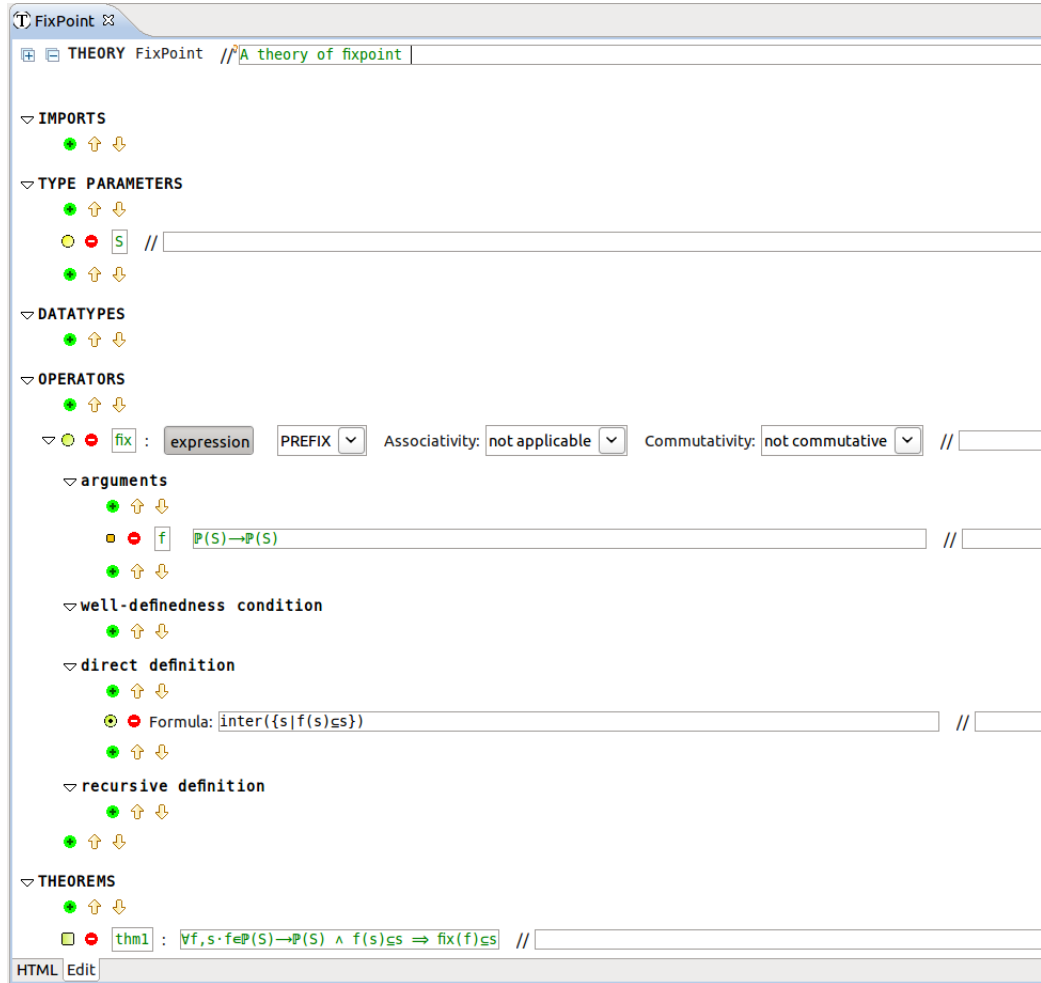


Figure 12.1: The Theory Component

- The specification of polymorphic theorems. This is a convenient addition to the Theory plug-in.

It is important to note that the Theory plug-in supports full polymorphism on all contributed extensions (i.e., proof and language extensions). An accurate description of the plug-in capabilities can be found in [MB10a].

Chapter 13

Model-Based Test Generation

*J. Bendisposto, I. Dinca, V. Kozyura, M. Leuschel, D. Plagge,
A. Stefanescu, W. Wei, S. Wiczorek*

13.1 Introduction

Model-based testing (MBT) is an approach from software engineering that uses formal models as basis for automatic generation of test cases. A test case is defined as a sequence of actions (or events, or triggers) together with corresponding test data that can be executed against a System Under Test (SUT). There are different types of test models that can be used for MBT, many of them being state-based models (e.g. UML state diagrams). In DEPLOY, we investigate a version of MBT using Event-B models as test models. This research work provides a new feature in the Rodin platform, complementing the existing theorem proving and model-checking capabilities. Thus, the main purpose of MBT for Rodin is to provide the deployment partners an MBT method together with a Rodin plug-in that allows generation of test cases satisfying different coverage criteria (e.g. covering of all events in a model or covering paths to a set of target global states). This includes the generation of appropriate test data that satisfies the guards of the single test steps.

In this chapter, we will describe a couple of methods of generating test cases out of Event-B models including their prototype implementation. For MBT using state-based models, test generation algorithms usually traverse the state space starting in an initial state trying to reach a certain coverage criteria (e.g. state coverage) collecting the execution paths in a test suite.

Event-B models do not have an explicit state space, but its state space is given by value of the variables and the state is changed by the execution of events that are enabled in that state. The ProB tool has a good grip of the state space, being able to explore it, visualize it, and verify various properties using model checking algorithms. Such model checking algorithms can be used to explore the state space of Event-B models using certain coverage criteria (e.g. event coverage) and thus generating test cases along the traversal. Moreover, the input data that allows to trigger the different events provides the test data associated with the test cases.

The following different methods are used for test case generation from Event-B models:

- *Using explicit model-checking:* First, model-checking algorithms described in the previous paragraph were implemented and applied to message choreography models from SAP. They work fine for models with data with a small finite range. However, in case of variables with a large range (e.g. integers), the known state space explosion problem creates difficulties, since the model-checker explores the state enumerating the many possible values of the variables. This required to consider different approaches as described below.
- *Using data abstraction and constraint solving:* To avoid the state space explosion due to the large bounds of the variables, the second approach ignores these values in the first step and uses the model-checker only to generate abstract test cases satisfying the coverage criteria. However, these paths may be infeasible in the concrete model due to the data constraints along the path. The solution is to represent the intermediate states of the path as existentially quantified variables. The whole path is then represented as a single predicate consisting of the guards and before-after predicates of its events. ProB's improved constraint solver is then used to validate the path feasibility and find appropriate data satisfying the constraints.
- *Using model learning:* Event-B models are essentially abstract state machines. However, their states are not given explicitly; instead they can be implicitly derived from the values of the model variables. Since the notion of state is at the heart of MBT, we provide a model-learning approach that uses the notion of cover automata to iteratively construct a subset of a state space together with an associated test suite. The iterative nature of the algorithm fits well with the notion of refinement from the Event-B method.

Once a test suite is generated, the test suite can be usually further optimised accruing to different criteria. We address this matter in this chapter as well.

The chapter is structured as follows. Sections 13.2-13.4 present the above three methods in details, i.e. theory, examples and experiments. Section 13.5 describe the test suite optimisation according to different criteria.

The following papers served as a basis for this chapter [WKR⁺09, IDS, Din11], so more information can be found there.

13.2 MBT using explicit model-checking

In this section we will describe how to apply model-checking techniques for test generation from Event-B models. We describe the method directly on an example from the business domain, namely SAP choreography models, called Message Choreography Models (MCM).

13.2.1 The industrial example

The service choreography modeling language MCM complements the structural information of the communicating components (e.g. service interface descriptions and message types) with information on the message exchange between them. A detailed discussion of the underlying concepts of MCM and how they support service development can be found in [WKR⁺09]. MCM consists of different model types each defining different aspects of service choreographies.

- **Global Choreography Model.** The global choreography model (GCM) is a labeled transition system which specifies a high-level view of the conversation between service components. Its purpose is to define every allowed sequence of observed messages.
- **Local Partner Model.** The local partner models (LPMs) specify the communication-relevant behavior for exactly one participating service component. Due to the design process of MCM, each LPM is a structural copy of the GCM with extra constraints on some of the local transitions, usually leading to the affected sending actions being deactivated.
- **Channel Model.** The channel model (CM) describes the characteristics of the communication channel on which messages are exchanged between the service components. These characteristics determine for

example whether messages sent by one component preserve their order during transmission and are formalized by the WS-RM standard.

In our running example, there are two service components, a buyer and a seller that negotiate a sales order. The buyer starts the communication by sending a Request message that will be answered with a Confirm by the seller. The buyer afterwards has the choice either to send a Cancel that rolls back the previous communication and allows to restart the negotiation or to send an Order that successfully concludes the ordering process. Because we assume a (reliable) communication channel that is not necessarily preserving the message order, it might be observed that a Cancel is delivered after a new negotiation process already started.

In the GCM at the top of Figure 1, the arrows labeled with an envelope depict the interactions Request, Confirm, Cancel, Order, and Cancel(deprecated) which are ordered with the help of the states Start, Request, Reserved, and Ordered. The states Ordered and Start are so-called target states (thus connected with the filled circle). Only in these states, the communication between the partners is allowed to terminate.

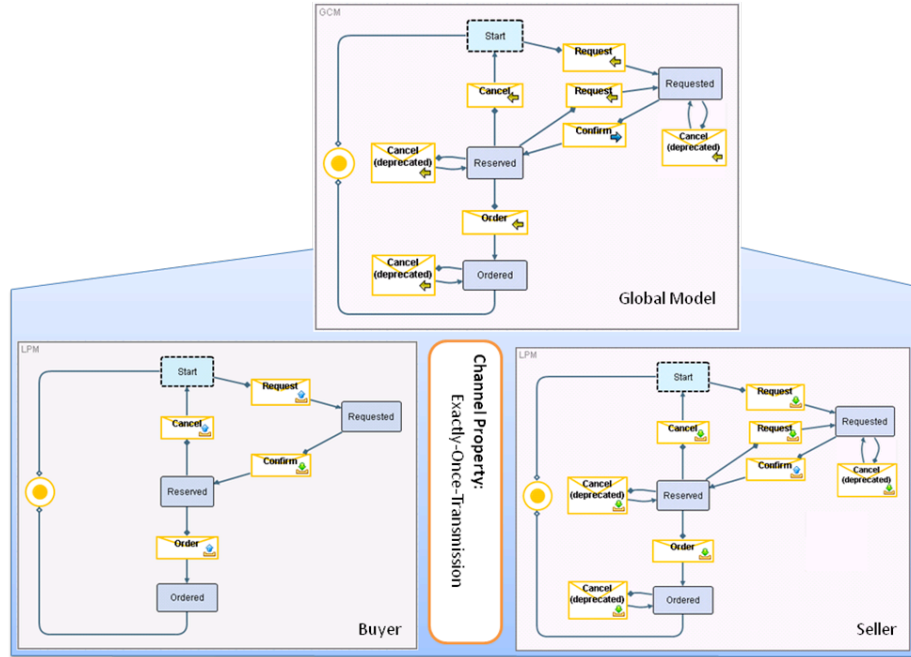


Figure 13.1: An MCM using global and local models

In order to generate tests via Event-B tooling, a transformation of MCM to Event-B is developed. We do not give the details here, but they can be

found in [WKR⁺09]

After having obtained a formal representation of the MCM model, we can employ a model checker to derive a test suite for integration testing. Similar to [24], we define integration testing as testing of an assembly of individually already tested components. Because of the confidence about the correctness of the participating components (which results from quality ensuring techniques on the component level, e.g. unit tests), our testing approach focuses on showing that each sent message is interpreted in the correct way by the receiver. This can be determined by checking for each interaction, that the intended message effect has been caused. Consequently, a test suite should cover all receive events modeled in the LPMs.

For automatic test generation, a local model that incorporates information from both LPMs and the CM (to connect the send and receive events) can be used. Because various cases studies show that state space explosion is the major stumbling point when applying automatic test generation to industrial settings, we decided to use the GCM to drive the test generation instead of the much more complex local model. While transition coverage of the GCM is equivalent to receive event coverage of the LPMs in most cases, the state space that needs to be explored is significantly lower.

Important from an industrial perspective is, that our approach further aims to be optimal regarding the minimization of the effort in the subsequent test concretization (e.g. provisioning of test data), test execution and test analysis phases. Based on practical experience of the testing process at SAP, we concluded that optimal corresponds to the following list of objectives which is sorted from highest to lowest priority:

1. Each path should start in the initial state and end in a target state: As described in [28] setting system states in test preambles is complicated and time consuming. Stopping a test while the system is not in a target state leads to problems with inconsistent data that might hamper consequent test executions.
2. The length of the longest generated path should be minimal: The longer a test case gets, the harder it is to maintain. Therefore especially for generated tests a top priority is to carefully control path lengths
3. Message racing should be minimal: Testing the effects that message racing has on the interaction is an important part of each test suite. Tests are mostly carried out in rather idealistic environments where messages are received in the same order they have been sent. Therefore, during test execution, message racing has to be emulated on the channel in a controlled way, usually leading to much higher effort.

4. The number of test steps should be minimal: As the effort increases with the overall length of all test cases, the sum of test steps should be minimized.

13.2.2 Test generation

In this subsection we describe how we utilize ProB to obtain an optimized test suite (regarding the objectives explained above) from the translated MCM models.

ProB is a validation toolset originally written for the B method. Its automated animation facilities allow users to animate and model-check their specifications which are valuable capabilities in the development of formal specifications. While consistency can be proven within tools such as Rodin or AtelierB, they are not capable of validating whether the model matches the specification that the modeler intended. Using the ProB animator, confidence in the models can be gained while using the model checker allows (at least for a part of the model's state space) to verify that a certain property holds. ProB has been adapted to support a number of formalisms such as Z, CSP, and CSP || B. Recently a ProB plug-in for the Rodin Platform has been developed, that can be used to animate and model check an Event-B specification within Rodin and to export Event-B models for using it in the ProB application. In the MCM editor the animation of the generated models is used but a detailed description here is out of scope.

The test generation algorithm we developed for the MBT approach based on MCM is separated into three steps. In the following we describe each step, give details about the implementation and show the computed results when applying it to the example above.

Step 1: Generation of the Initial Global Test Suite. As explained, our aim is to cover each transition of the global communication model, i.e. each interaction of the GCM. As each interaction is translated into a separate Event-B event, we have to ensure that every event is covered by at least one concrete transition in the state space of the global Event-B model, from which a valid end state can be reached. Note that the same event is typically covered by many different transitions, as its parameters can be valued in many different ways. In our particular example, the full state space is actually infinite, due to the use of integers as message identifiers. In order to reduce the state space, we have to configure ProB to compute only a few possible

ways to enable any event¹.

To satisfy the first and second objective given in previous subsection, we have extended ProB to detect when full transition coverage is obtained². This is gained by exploring the state space of the model breadth first, stopping when full coverage is achieved. Note that we also need to secure that for every operation we can reach a valid end state. This has been ensured by refining the Event-B translation, by adding a history variable, storing the set of executed events, and adding a corresponding end-event for every original event e which can be triggered if we are in a valid end state and if $e \in \text{history}$. Afterwards all traces that end in a target state are extracted from the explored state space to form the initial test suite. From the example in this chapter, we obtain the following initial test suite:

```
[Request, Confirm, Order], [Request, Confirm, Cancel],
[Request, Confirm, Cancel, Request, Confirm, Order],
[Request, Confirm, Cancel, Request, Confirm, Cancel],
[Request, Confirm, Request, Confirm, Order],
[Request, Confirm, Request, Confirm, Order, Cancel(depr.)],
[Request, Confirm, Request, Confirm, Cancel],
[Request, Confirm, Request, Confirm, Cancel(depr.), Order],
[Request, Confirm, Request, Confirm, Cancel(depr.), Cancel],
[Request, Confirm, Request, Cancel(depr.), Confirm, Order],
[Request, Confirm, Request, Cancel(depr.), Confirm, Cancel]
```

The computation takes 0.32 seconds on a 2.33 GHz Core2 Duo laptop and should scale up to much larger examples.

Step 2: Mapping of Global to Local Paths. In order to obtain executable test cases the global sequence of message observations for each path has to be mapped to the corresponding send and receive events of partners. As the GCM uses receive semantics, the global observe sequences can be directly translated to sequences of receive events. In the case of the path

```
[Request, Confirm, Request, Confirm, Cancel(depr.), Cancel]
```

the resulting sequence is (? reads “receives”):

```
[Seller?Request, Buyer?Confirm, Seller?Request, Buyer?Confirm, Seller?Cancel(depr.),
Seller?Cancel]
```

Afterwards for each receive event a corresponding send event is generated and added to the path in such a way that the local behavior descrip-

¹This approach has proven to be sufficient so far, but in future, we will consider using ProB’s symmetry reduction instead.

²Note that this is a property that cannot be expressed as an LTL formula, as it is not a property of individual paths but of the entire state space explored so far.

tions are not violated. In the mentioned sequence the send event for `Cancel(deprecated)` has to be added before the second `Request` as the Buyer is not able to send these messages in the same order as they have to be received for the test. The resulting local sequence from our example therefore is (! reads sends):

[Buyer!Request, Seller?Request, Seller!Confirm, Buyer?Confirm, Buyer!Cancel, Buyer!Request, Seller?Request, Seller!Confirm, Buyer?Confirm, Seller?Cancel(depr.), Buyer!Cancel, Seller?Cancel]

The message racing in the illustrated local path is underlined. While the `Cancel` message is sent by the buyer before the `Request` message, the seller receives the `Request` message first.

Similar to Step 1, it is again infeasible to exhaustively explore the full state space (as the state space of the local model is actually even considerably bigger) to find a suitable mapping from global to local traces. One could encode the problem as an LTL formula, but this formula will be very big with ensuing consequences for the complexity of model checking. The solution we have come up with, is to encode the desired LCM scenarios into a CSP process. This process is synchronized with the Event-B model, suitably guiding the model checker. The CSP Process is divided into two components.

The first process encodes the desired trace of receive events, followed by an event on the goal channel, indicating to the model checker that this is a goal state we are looking for. For the trace given above it looks as follows:

RECEIVER = Seller?Request $\longrightarrow$ Buyer?Confirm $\longrightarrow$ Seller?Request $\longrightarrow$ Buyer?Confirm $\longrightarrow$ Seller?Cancel(depr.) $\longrightarrow$ Seller?Cancel $\longrightarrow$ goal $\longrightarrow$ STOP

The second process encodes the sender events. We know how many send events of each type must occur, but the order of these is unknown.

$SENDER(n1, n2, n3, n4) =$

$n1 > 0 \& Buyer!Request \longrightarrow SENDER(n1 - 1, n2, n3, n4)[]$

$n2 > 0 \& Seller!Confirm \longrightarrow SENDER(n1, n2 - 1, n3, n4)[]$

$n3 > 0 \& Buyer!Cancel \longrightarrow SENDER(n1, n2, n3 - 1, n4)[]$

$n4 > 0 \& Buyer!Order \longrightarrow SENDER(n1, n2, n3, n4 - 1)$

The sender process is now simply interleaved with the receiver process.

MAIN = SENDER(2,2,2,0) ||| RECEIVER

Now, ProB will ensure that every event of the Event-B model synchronizes with an event of the CSP process (MAIN) guiding it and stopping when the CSP process can perform an event on the goal channel. For the initial test suite from Step 1, we compute a described mapping for each global trace in

0.064 seconds.

Step 3: Test Suite Reduction. The resulting test suite incorporating the local traces is now ready to be optimized according to the third and fourth objective from Section 2.3. The optimization of the test suite and the test suite reduction has been implemented in Java. In the first prototypical version we use a brute force algorithm that computes every possible combination of test cases and selects the optimal one according to the given objectives. The computed optimal test suite incorporates the local equivalents of the following global paths:

[Request, Confirm, Request, Cancel(depr.), Confirm, Order],
[Request, Confirm, Request, Confirm, Cancel(depr.), Cancel],
[Request, Confirm, Request, Confirm, Order, Cancel(depr.)]

For the given example the test suite is produced in less than a millisecond, implying that it is applicable in practice. However as the algorithms computational complexity is exponential in the number of test cases of the extended suite, we are planning to apply the following more sophisticated approach that reduces the number of computations: First it is analyzed which of the global interactions can only be covered by paths incorporating message racing. In our example these are the three interactions called Cancel (deprecated). For these a minimal set of covering paths is determined using a greedy algorithm. If more than one possibility exists, the one that has the highest overall interaction coverage is chosen. The resulting test suite is filled with the minimum set of paths (not incorporating message racing) that covers the remaining interactions.

More test suite reductions algorithms and minimization criteria are provided in Section 13.5.

13.3 MBT using constraint solving

Using the explicit model-checking approach exposes a problem that we illustrate with a simple example of a bank account. We have just one variable *balance*, initialised with 0, and are only interested in the two events *deposit* and *checkGold*.

deposit is used to add a certain positive amount *a* to the balance:

deposit $\hat{=}$ ANY *a* WHERE *a* > 0 THEN *amount* := *amount* + *a* END

checkGold is used to validate if the account contains a minimal amount of money. It does not change the state:

checkGold $\hat{=}$ WHEN *a* $\geq$ 1000 THEN skip END

With the explicit model-checking approach we do not find a test case that covers the event *checkGold*. The reason is that ProB explores only a limited number of solutions for a single event in one state. E.g. in the initial state where *amount* is 0, we have theoretically an infinite number of possibilities for choosing *deposit*'s parameter *a*, each choice leading to a state with a different amount. To keep the problem tractable, ProB stops after finding a fixed number of solutions. The default maximum is 10 solutions per state and event. Thus, with one *deposit* event, ProB finds only the 10 new states where the amount is between 1 and 10 and 100 *deposit* events must take place before *checkGold* would be enabled. The maximum length for test cases is usually much smaller.

A naive approach to this problem would be to raise the maximum number of computed solutions or the maximum length of the test cases. But with more complex models, it is not necessarily clear which parameter to set. And more importantly, those approaches do not scale. The generated state space usually will become very large.

13.3.1 Constraint solving with ProB

To see how ProB's constraint solving facility can help and why it is not applied in the model checking based algorithm of Section 13.2, we have a more detailed view at the constraint solving process. Driven by industrial need [LFFP09], ProB's ability to find solutions to B and Event-B predicates has been significantly improved during the Deploy project.

To illustrate the constraint solving process, we take a simple example with two variables x, y and the predicates $x = y + 5$, $y \in 5..7$, $\exists z \cdot z * 2 = y$. After processing $x = y + 5$, ProB just stores the relation between x and y . After the second predicate, we know $10 \leq x \leq 12$ and $5 \leq y \leq 7$. The third predicate just adds a "passive" constraint that does not directly add new information but waits for values of y to check.

To find concrete solutions for the problem, ProB starts to enumerate a variable. Let's assume that x is taken. Then ProB tries values between 10 and 12, so it makes use of the inferred information from the second predicate. For $x = 10$, the second constraint leads to $y = 5$ and the last constraint is triggered and fails, so the solution is rejected. For $x = 11$, the second constraint leads to $y = 6$ and the last constraints evaluates to true.

The constraint solver has finished when all variables have concrete values and all constraint are evaluated. When exploring the state space, ProB takes an existing state and uses the constraint solver to find a number of new states for each event. A new state consists of concrete values for all variables. The constraint solving process is applied separately and for each event, thus the

guard $amount \geq 1000$ cannot influence the search for suitable values for *deposit*'s parameter a .

The basic idea of the approach presented here is to set up a sequence of events as a single constraint solving problem and try to find values for all intermediate values. Let's see how such a constraint problem would look like for the sequence $[INITIALISATION, deposit, checkGold]$. We use $amount$ for the initial state, $amount'$ for the intermediate state between *deposit* and *checkGold* and $amount''$ for the final state. Then we have the following constraints:

$amount = 0$	from <i>INITIALISATION</i>
$amount' = amount + a \wedge a > 0$	from <i>deposit</i>
$amount' \geq 1000 \wedge amount' = amount''$	from <i>checkGold</i>

After processing all constraints we know $amount = 0$, that $amount'$, a and $amount''$ are the same and $amount' \geq 1000$. To find a solution ProB needs to enumerate only one variable and has the information that it can start with the value 1000. So it would find a solution immediately.

13.3.2 The test case generation algorithm

To generate test cases systematically, we need a procedure to set up sequences of events. For each sequence we call ProB's constraint solver as described above. The algorithm only tries to find a sequence $p \leftarrow e$ ($p \leftarrow e$ denotes the sequence with the prefix p and one appended element e) only if a solution for p was found.

The detailed algorithm can be seen in figure 13.2. The following variables are used in the procedure:

- *cur_length* is the length of the event sequence in the current iteration.
- *paths* contains the feasible paths found in the last iteration. Implicitly every path is preceded by the initialisation event.
- *testcases* contains the set of all found test cases.
- *new_paths* contains the feasible paths found in the current iteration.

The keyword **solve** indicates a call to the constraint solver that returns a solution if one found. Each call to the constraint solver is interrupted after a fixed time because we want to avoid that complicated combinations of constraints block the whole search for test cases.

A test case is only valid if the sequence ends in a state where the target predicate is true. First we try for each sequence of events to solve the constraints together with the target predicate applied to the last state. If we find a solution we have encountered a new test case. Otherwise we try to solve the constraints of the sequence without the target predicate. This is necessary to check if the sequence can serve as a prefix for other test cases. If the target predicate is simply true, the second call to the constraint solver without the predicate can be omitted.

The procedure stops after if all events are covered or the maximum length of test cases is reached. In the latter case, the found test cases do not cover all events.

A requirement of the industrial partner for the test case generation algorithm is that when exploring test cases of a certain length n , all possible test cases of length n should be generated. The rational for this requirement is that another tool should optimize the test suite (the set of applied test cases) by choosing an appropriate subset. For this reason the algorithm has a delayed termination, the currently explored length is always fully explored.

13.3.3 Case studies

We applied the algorithm to a number of case studies to see if it is applicable and where the method could be improved.

All experiments were carried out on a 2.8 Mhz Intel i7 processor running under Linux. and are executed with a maximum of 2 seconds of computation time for each path.

Table 13.1 gives an overview about the case studies. The columns 1–8 refer to the length of the path for which are solutions search for. In the algorithm in Figure 13.2 that would be the variable *cur_length*. For each model we show

- the number of paths set up for a certain depth length (this is the number of iterations of the **for**-loop in Figure 13.2 for each value of *cur_length*),
- the number of paths where a solution was found (found solutions s_1 for each *cur_length*),
- the number of paths where a solution was found that was also a test case (found solutions s_2 for each *cur_length*) and
- the time in milliseconds that was needed to compute the test cases.

We can see that the number of a checked paths of length n is exactly the number of found paths of length $n - 1$ multiplied with the number of events in the model.

Generally, the runtime and number of generated test cases can change drastically due to small changes in the model. E.g. if there is one event in a model that is never enabled due to an erroneous model, the search algorithm searches until the maximum defined length, even if all other events are covered after a few steps.

Bank account We already used the bank account example above to illustrate the advances of the constraint based approach in comparison to the model checking approach. All generated test cases must end in a state where the balance is 0.

The example contains one event *evaluateRed* that can be only enabled under certain conditions. In table 13.1, we show the numbers for to runs of the algorithm. First, we let ProB compute the test cases without trying to cover *evaluateRed*. In that case all other events are covered within 30 milliseconds. But when adding *evaluateRed*, the algorithm needs 85 seconds for generating all test cases. This illustrates that small changes in the model can lead to a completely different run time.

Basket The Basket is another model that was written as an example for test case generation. It models a shopping basket where several items can be added to. The model is not large but it exposed a limitation of the constraint solving algorithm dealing with the cardinality of sets. One event is only enabled when the size of the shopping card has reached a minimum. This information is currently not actively used when finding solutions and a solution was only found by ProB when enumerating possible solutions.

SAP Choreography Model The model is based on a choreography model by SAP. It represents a simple state machine without more complex data. A variable contains the current state. The resulting state space is shown in Figure 13.3. This model is an example where the constraint based search is applicable but much slower than the model checking approach which generates the test cases in just 30 milliseconds.

Cruise Control The cruise control system is an industrial example, written in classical B. It has 26 operations, the model's state space is finite and consists of 1361 states. The following table shows the results when generating test cases for all specified operations. 5 steps were necessary to cover all events.

Without a target predicate, ProB needed 413 seconds to generate all 22,488 test cases. The large number of test cases is due to the fact that many of the 26 operations are enabled in each state and so the number of possibilities for possible pathes grows rapidly. Without a target predicate, each solution to a path is also a found test case.

A second run where a target predicate was given (the cruise control had to be turned off in the final state), 360 seconds were needed to generate 15,551 test cases. One operation has not been covered but we decided to limit the algorithm to the maximum length 5.

The cruise control system exhibits an inherent problem with the breadth-first search: When many operations are enabled, the number of paths that must be checked can become very large. ProB had to check 68,822 paths of length 5 whether it can find a solution, in total it had to check 81,068 paths.

Future work should investigate how the number of paths could be minimized and the requirement that all paths of a certain length should be computed must be weakened for some models.

Quicksort In [Hal09b] an imperative implementation of the Quicksort algorithm was developed and proved correct by refinement in Event-B. We have taken the model's ninth refinement that contains 8 different events. The algorithm's input is a sequence of numbers.

It did not find solutions to cover all events when the input was completely unrestricted, because the computations were interrupted after 2 seconds.

After restricting the input to have three elements, ProB found test cases that did cover all events.

Greatest Common Divisor The Greatest common divisor (taken from [Hal10]) is another example for finding test cases for an imperative algorithm modelled with Event-B. It turned out that the one test case was enough to cover all events.

13.3.4 Conclusion

The test case generation as described above has been implemented and used as a web service in SAP's infrastructure to generate test cases for their choreography models.

The case studies indicate that the method is principally applicable for many models. One problem was that our approach currently tries to find all possible test cases of a certain length. The reason for this is that the optimization of a test suite should be applied in a later step independently of ProB. The downside is that for some larger models the number of possible test paths can become very large such that the method does not scale. To make the method applicable for a wider range of models, it is interesting to weaken the requirement of generating all test cases which enables us to lessen the number of paths that are given to the constraint solver. There are several possibilities to do that:

- Heuristics can be applied to select a subset of the paths.
- Certain events that do not change the state or do not affect some variables could be sorted out.
- The flow analysis ([BL11]) can be used to infer statically which events can be combined under certain conditions.

Procedure CBTestCaseGeneration

Input: The events to cover to_cover ,
and the maximum test case length max_length .

\— *Initialisation* —\
 $paths := \{[]\}$ — $[]$ is the path of length 0
 $testcases := \emptyset$
 $cur\_length := 1$
\— *Breadth first search* —\
while $cur\_length \leq max\_length \wedge to\_cover \neq \emptyset$ **do**
 $next\_paths := \emptyset$
 for each path $p \in paths$ and event e **do**
 $s_1 := \textbf{solve}$ constraints of the path $p \leftarrow e$
 where its last state fulfills the target predicate.
 if $s_1 \neq fail$ **then**
 $next\_paths := next\_paths \cup \{p \leftarrow e\}$
 $testcases := testcases \cup \{s_1\}$
 \— *With $p_1, \dots, p_{cur\_length-1}$ beeing the events in p :* —\
 $to\_cover := to\_cover \setminus \{p_1, \dots, p_{cur\_length-1}, e\}$
 else
 $s_2 := \textbf{solve}$ constraints for the path $p \leftarrow e$
 if $s_2 \neq fail$ **then**
 $next\_paths := next\_paths \cup \{p \leftarrow e\}$
 end for
 $paths := next\_paths$
 $cur\_length := cur\_length + 1$
end

Figure 13.2: Constraint based test case generation

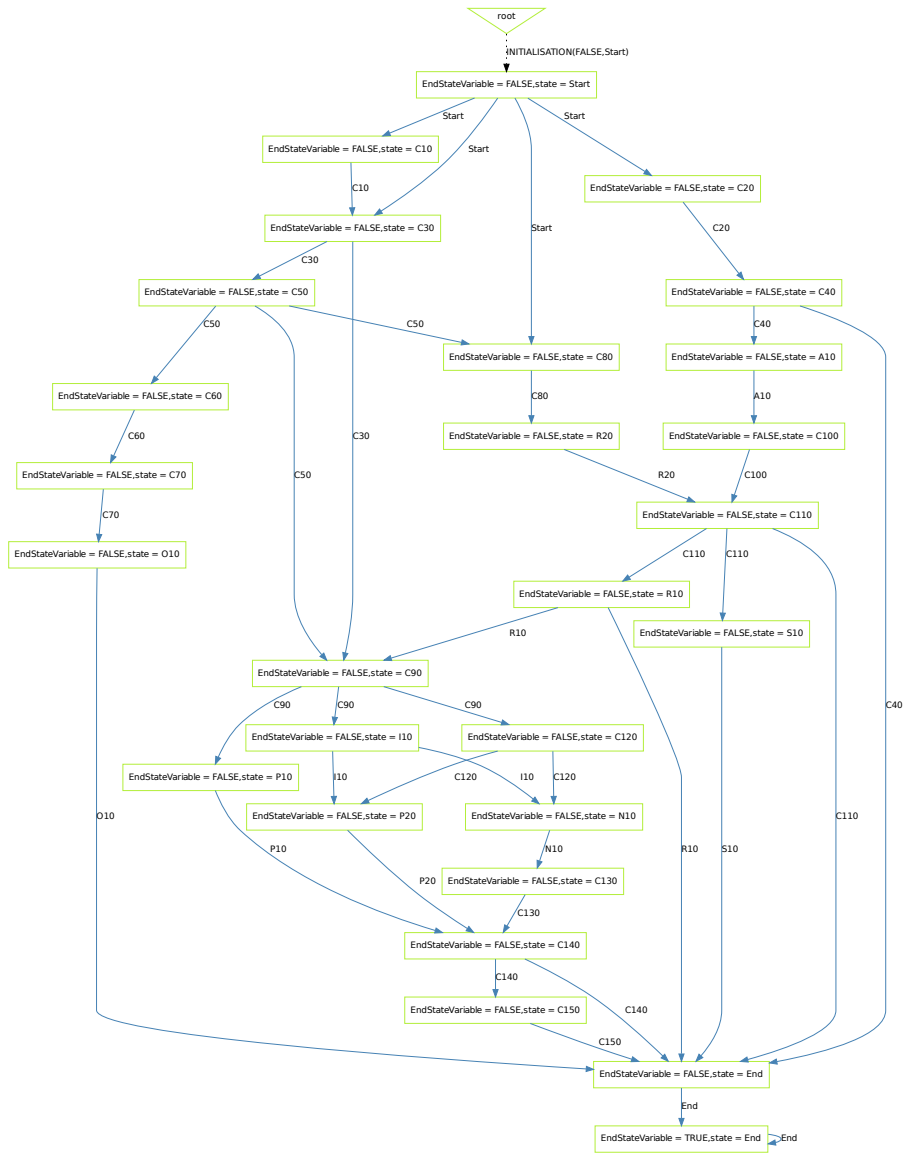


Figure 13.3: State space of the choreography model

	1	2	3	4	5	6	7	8	Total Time [ms]	
Bank Account										
Paths	8	16	64	320	1.584				408 ¹ /1992 ²	
Solutions	2	8	40	198	958				248 ¹ /1206 ²	
Test cases ¹	1	2	8	37					48	30
Test cases ²	1	2	8	37	168				216	84.860
Basket										
Paths	6	12	30	90	288				426	
Solutions	2	5	15	48	156				226	
Timeouts				3	23				26	
Test cases	2	5	15	48	156				226	55.530
SAP Choreography Model										
Paths	26	26	104	130	286	546	754	1066	2.938	
Solutions	1	4	5	11	21	29	41	54	166	
Test cases				1	2	5	13	30	51	10.480
Cruise Control										
Paths	26	234	1.612	10.374	68.822				81.068	
Solutions	9	62	399	2.647	19.371				22.488	
Test cases ³	9	62	399	2.647	19.371				22.488	413.160
Test cases ⁴	9	61	369	2.159	12.953				15.551	346.090
Quicksort										
Paths	8	8	24	72	152	152	152	200	768	
Solutions	1	3	9	19	19	19	25	37	132	
Test cases								13	13	13.300
Greatest Common Divisor										
Paths	1	5	5	5	5				25	
Solutions	1	1	1	1	1				5	
Test cases					1				1	2.950

¹: without covering *validateRed*

²: with *validateRed* covered

³: without target predicate

⁴: with target predicate

Table 13.1: Evaluation of the test case generation

13.4 MBT using model learning

The concept of state is at the heart of model-based testing and many test generation techniques from finite state machines (FSMs) exist. However, FSMs are not powerful enough to efficiently model realistic systems and so extended finite state machines (EFSMs), such as Statecharts, are used instead; these combine a FSM-like control with suitable data variables and operations for these variables, to offer an intuitive, yet rigorous means for system modeling and analysis. Testing from an EFSM usually involves transforming the EFSM into an equivalent FSM (whose states are given by the state-variable value combinations of the original EFSM) and then applying FSM-based test generation techniques. However, for many systems, the equivalent FSM may have many more states than the length of the tests that can realistically be performed, or, furthermore, the number of states of the resulting FSM may be so large that it is impossible to even construct it. This is the well-known state explosion problem. Despite the existence of numerous techniques for alleviating this problem, state explosion remains one of the major obstacles for efficient model-based test generation from state-based models.

In this section we propose an approach which, given a state-transition model (EFSM) of a system, constructs, in parallel, an *approximate FSM* model and a *test suite* for the system. The (approximate) model construction relies on a variant of Angluin’s automata learning algorithm [Ang87], adapted to finite cover automata [Ipa11]. A *finite cover automaton* [CPS06] of a finite set L is a finite automaton which accepts all sequences in L but may also accept sequences that are longer than every sequence in L . The main advantage of a finite cover automaton is that its size (number of states), which normally depends on ℓ , can be significantly lower than the size of the automaton which accepts exactly the language L [CPS06]. In practice, an upper bound ℓ on the length of the considered sequences will be established and the constructed model will have to conform to the original model for all sequences of length at most ℓ . In this way, by appropriately setting the value of the bound ℓ , the state explosion problem normally associated with constructing and checking state based models can be addressed. Furthermore, test generation from finite cover automata fits very well with common testing practices, that usually require test cases of short to medium length and can also be regarded as a natural complement to Bounded Model Checking (BMC) based on SAT methods [PBG05], which is gaining popularity in the formal verification community.

The proposed approach also allows for a gradual construction of the model and of the associated test suite: the FSM model and test suite for an initial version of the system are reused in the construction of a more elaborated and

complex version, with complexity and time savings, but also with improvements in the accurateness of the obtained models and tests.

The approach is presented and implemented for the Event-B modeling language, but its underlying ideas and principles are much more general and can be applied to any system whose behavior can be suitably described by a state-transition model. As Event-B does not even distinguish between state and data variables, such models are very suitable means for evaluating our approach. ¹

Given an Event-B model and an upper bound ℓ , the proposed approach will incrementally construct finite cover automata that will eventually accept all executable sequences of length less than or equal to ℓ . As a by-product of the automata learning algorithm, a *set of test cases* associated with the cover automata is also maintained and evolved during the iterations. This test suite can be used for conformance testing of the modeled system. The test cases in the test suite are provided together with the associated *test data* that makes them executable on the Event-B model.

The contributions of our approach are threefold:

- first, it constructs a successive set of finite approximation models for the set of Event-B executable traces up to a length ℓ . The construction exploits the restriction given by the bound ℓ to obtain models of reduced size (number of states) compared to exact automaton models. Moreover, the cover automata are minimal by construction.
- second, in parallel with automata construction, we incrementally generate conformance test suites for the investigated Event-B models. By construction, the generated test cases satisfy certain minimality properties regarding their lengths. This fits very well with the testing practice that usually requires short test cases.
- third, the Event-B method deploys model refinement as a means to handle modeling complexity. The two contributions above can be applied incrementally, allowing the reuse of the learned model and test cases from the abstract to the more concrete levels.

The section is structured as follows. Subsection 13.4.1 recalls theoretical foundations, including the used algorithm for the cover automata. Subsections 13.4.3 and 13.4.5 describe the adaptation to Event-B and its implementation, respectively. Subsection 13.4.6 concludes the section.

13.4.1 Theoretical background

This subsection, which is largely adapted from our previous work [Ipa11], presents the L^ℓ algorithm and its automata-related concepts.

Before continuing, we introduce the notations used in the section. For a finite alphabet A , A^* denotes the set of all finite sequences with members in A . ϵ denotes the empty sequence. For a sequence $a \in A^*$, $\|a\|$ denotes the length (number of symbols) of a ; in particular $\|\epsilon\| = 0$. For a finite set of sequences $U \subseteq A^*$, $\|U\|$ denotes the length of the longest sequence(s) in U . For $a, b \in A^*$, ab denotes the concatenation of sequences a and b . a^n is defined by $a^0 = \epsilon$ and $a^n = a^{n-1}a$, $n \geq 1$. For $U, V \subseteq A^*$, $UV = \{ab \mid a \in U, b \in V\}$; U^n is defined by $U^0 = \{\epsilon\}$ and $U^n = U^{n-1}U$, $n \geq 1$. $A[n] = \bigcup_{0 \leq i \leq n} A^i$ denotes the sets of sequences of length less than or equal to n with members in the alphabet A . For a sequence $a \in A^*$, $b \in A^*$ is said to be a *prefix* of a if there exists a sequence $c \in A^*$ such that $a = bc$. The set of all prefixes of a is denoted by $\text{pref}(a)$; for $U \subseteq A^*$, $\text{pref}(U) = \bigcup_{a \in U} \text{pref}(a)$. For a sequence $a \in A^*$, $b \in A^*$ is said to be a *suffix* of a if there exists a sequence $c \in A^*$ such that $a = cb$. For a finite set A , $\text{card}(A)$ denotes the number of elements in A .

Finite automata - general concepts

We start by introducing some classic definitions from automata theory.

A *deterministic finite automaton* (DFA) M is a tuple (A, Q, q_0, F, h) , where: A is the finite *input alphabet*; Q is the finite *set of states*; $q_0 \in Q$ is the *initial state*; $F \subseteq Q$ is the *set of final states*; h is the *next-state*, $h : Q \times A \rightarrow Q$. A DFA is usually described by a *state-transition diagram*.

The next-state function h can be naturally extended to a function $h : Q \times A^* \rightarrow Q$. A state $q \in Q$ is called *reachable* if there exists $s \in A^*$ such that $h(q_0, s) = q$. M is called *reachable* if all states of M are reachable.

Given $q \in Q$, the set L_M^q is defined by $L_M^q = \{s \in A^* \mid h(q, s) \in F\}$. When q is the initial state of M , the set is called the *language accepted by M* and the simpler notation L_M is used. Given $Y \subseteq A^*$, two states $q_1, q_2 \in Q$ are called *Y -equivalent* if $L_M^{q_1} \cap Y = L_M^{q_2} \cap Y$. Otherwise q_1 and q_2 are called *Y -distinguishable*. If $Y = A^*$ then q_1 and q_2 are simply called *equivalent* or *distinguishable*, respectively. Two DFAs are called *(Y -)equivalent* or *(Y -)distinguishable* if their initial states are (Y -)equivalent or (Y -)distinguishable, respectively.

A DFA M is called *reduced* if every two distinct states of M are distinguishable. A DFA M is called *minimal* if any DFA that accepts L_M has at least the same number of states as M . A DFA M is minimal if and only if M

is reachable and reduced. Furthermore, there is an unique (up to a renaming of the state space) minimal DFA that accepts a given regular language.

Now let us also introduce the concept of *deterministic finite cover automaton* (DFCA). Informally, a DFCA of a finite language U , as defined by Câmpeanu et al. [CPS06], is a DFA that accepts all sequences in U and possibly other sequences that are longer than any sequence in U .

In this section we use a slightly more general concept, as defined in [Ipa11]: given a finite language $U \subseteq A^*$ and a positive integer ℓ that is greater than or equal to the length of the longest sequence(s) in U , a *deterministic finite cover automaton* (DFCA) of U w.r.t. ℓ is a DFA M that accepts all sequences in U and possibly other sequences that are longer than ℓ , i.e. $L_M \cap A[\ell] = U$. A DFCA M of U w.r.t. ℓ is called *minimal* if any DFCA of U w.r.t. ℓ has at least the same number of states as M . Note that, unlike the case in which the acceptance of the exact language is required, the minimal DFCA is not necessarily unique (up to a renaming of the state space) [Ipa11].

Naturally, a DFA that accepts a finite language U is also a DFCA of U w.r.t. any $\ell \geq \|U\|$. Consequently, the number of states of a minimal DFCA of U w.r.t. ℓ will not exceed the number of states of the minimal DFA accepting U . Furthermore (and more importantly from the point of view of practical applications), the size of a minimal DFCA of U w.r.t. ℓ can be much smaller than the size of the minimal DFA that accepts U [Ipa11].

The L^ℓ algorithm for learning finite cover automata

Learning regular languages from queries was introduced by Angluin in [Ang87]; the paper also provides a learning algorithm, called L^* . The L^* algorithm infers a regular language, in the form of a DFA from the answers to a finite set of membership queries and equivalence queries. A *membership query* asks whether a certain input sequence is accepted by the system under test or not. In addition to membership queries, L^* uses *equivalence queries* to check whether the learning algorithm is completed.

In a recent paper [Ipa11], we extended Angluin's work by proposing an algorithm, called L^ℓ , for learning a DFCA. Given an unknown finite set U and a known integer ℓ that is greater than or equal to the length of the longest sequence(s) in U , the L^ℓ algorithm will construct a minimal DFCA of U w.r.t. ℓ . Analogously to L^* , the L^ℓ algorithm uses membership and language equivalence queries to find the automaton in polynomial time.

The L^ℓ algorithm construct two sets: S , a non-empty, prefix-closed set of sequences and W , a non-empty, suffix-closed set of sequences. Additionally, S will not contain sequences longer than ℓ and W will not contain sequences longer than $\ell - 1$, i.e. $S \subseteq A[\ell]$ and $W \subseteq A[\ell - 1]$.

The algorithm keeps an *observation table*, which is a mapping T from a set of finite sequences to $\{0, 1, -1\}$. The sequences in the table are formed by concatenating each sequence of length at most ℓ from the set $S \cup SA$ with each sequence from the set W . Thus, the table can be represented by a two-dimensional array with rows labeled by elements of $(S \cup SA) \cap A[\ell]$ and columns labeled by elements of W .

The function $T : ((S \cup SA) \cap A[\ell])W \rightarrow \{0, 1, -1\}$ is defined by $T(u) = 1$ if $u \in U$, $T(u) = 0$ if $u \in A[\ell] \setminus U$ and $T(u) = -1$ if $u \notin A[\ell]$. The values 0 and 1, respectively, are used to indicate whether a sequence is contained in U or not. However, only sequences of length less than or equal to ℓ are of interest. For the others, an extra value, -1 , is used.

In order to compare the rows in the observation table, a relation on these rows, called *similarity*, is used. We say that rows s and t are k -similar, $1 \leq k \leq \ell$, and write $s \sim_k t$ if, for every $w \in W$ with $\|w\| \leq k - \max\{\|s\|, \|t\|\}$, $T(sw) = T(tw)$. Otherwise, s and t are said to be k -dissimilar, written $s \not\sim_k t$. In other words, the table values of rows s and t must coincide for every column w for which the lengths of sw and tw are both less than or equal to k . The relation $\sim_k$ is not an equivalence relation since it is not transitive [Ipa11]. When $k = \ell$, we simply say that s and t are *similar* or *dissimilar* and write $s \sim t$ or $s \not\sim t$, respectively. It can be observed that similarity of rows s and t requires all corresponding non-negative values of the two rows to coincide.

Using the similarity relation, two properties of an observation table are defined: consistency and closedness.

The observation table is *consistent* if, for every k , $1 \leq k \leq \ell$, whenever rows $s_1 \in S$ and $s_2 \in S$ are k -similar, rows s_1a and s_2a are also k -similar for all $a \in A$.

The observation table is *closed* if, for all rows $s \in SA$, there exists row $t \in S$ with $\|t\| \leq \|s\|$, such that $s \sim t$.

Consider, for example, $A = \{a, b\}$, $\ell = 3$ and Table 13.2 (left hand side) - in which a double horizontal line is used to separate the rows labeled with elements of S from the rows labeled with elements of $SA \setminus S$ - to be the current observation table ($S = \{\epsilon, a, b, aa, bb\}$, $W = \{\epsilon, a\}$). The observation table is not consistent since, for $k = 2$, $s_1 = \epsilon$, $s_2 = b$, $w = \epsilon$ and $\alpha = b$ satisfy $s_1 \sim_k s_2$, but $T(s_1\alpha w) \neq T(s_2\alpha w)$. On the other hand, the observation table is closed.

The algorithm starts with $S = W = \{\epsilon\}$. It periodically checks the consistency and closedness properties and extends the table accordingly. When both conditions are met, the DFA $M(S, W, T)$ corresponding to the table is constructed (details will be provided later on) and it is checked whether the language L accepted by $M(S, W, T)$ satisfies $L \cap A[\ell] = U$ (this is called a “lan-

T	ϵ	a	
ϵ	0	0	
a	0	1	
b	0	0	
aa	1	0	
bb	1	0	
ab	0	0	
ba	0	0	
aaa	0	-1	
aab	0	-1	
bba	0	-1	
bbb	0	-1	

T	ϵ	a	b
ϵ	0	0	0
a	0	1	0
b	0	0	1
aa	1	0	0
bb	1	0	0
ab	0	0	0
ba	0	0	1
aaa	0	-1	-1
aab	0	-1	-1
bba	0	-1	-1
bbb	0	-1	-1

Table 13.2: Observation table of running example (left table) and its updated form that is consistent and closed (right table)

guage query”). If the language query fails, a counterexample t is produced, the table is expanded to include t and all its prefixes and the consistency and closedness checks are performed once more. Eventually, the language query will succeed and the algorithm will return a minimal DFCA of U w.r.t. ℓ .

Since in our approach we will separate the construction of the observation table and of the corresponding DFCA (which is the actual processing performed by the algorithm) from the language queries (which represent the user intervention), only the processing performed between two language queries is presented in pseudo-code in Fig. 13.4 (in what follows this will be referred to as the *LearnDFCA* procedure).

The *LearnDFCA* procedure starts with the current values of S , W and the current observation table T . It periodically checks whether the consistency and closedness properties are violated and extends the table by adding a new row or a new column to the table, respectively:

- In order to check consistency, the procedure will search for $w \in W$ and $a \in A$ such that aw will distinguish between two rows s_1 and s_2 that are not distinguished by any sequences in W of length less than or equal to aw ; in order to find the shortest such sequence aw , the search will be performed in increasing order of length of w . The search is repeated until all elements of W have been processed; as these are processed in increasing order of their length, any sequence aw that has

Procedure LearnDFCA

Input: S , W and the current observation table T .

Repeat

 \— *Check consistency* —\

For every $w \in W$, in increasing order of $\|w\| = i$ **do**

 Search for $s_1, s_2 \in S$ with $\|s_1\|, \|s_2\| \leq \ell - i - 1$

 and $a \in A$ such that $s_1 \sim_k s_2$, where

$k = \max\{\|s_1\|, \|s_2\|\} + i + 1$, and $T(s_1aw) \neq T(s_2aw)$.

If found **then**

 Add aw to W .

 Extend T to $(S \cup SA)W$ using membership queries.

 \— *Check closedness* —\

 Set $new\_row\_added = false$.

Repeat for every $s \in S$, in increasing order of $\|s\|$

 Search $a \in A$ such that $sa \not\sim t \forall t \in S$ with $\|t\| \leq \|sa\|$.

If found **then**

 Add sa to S .

 Extend T to $(S \cup SA)W$ using membership queries.

 Set $new\_row\_added = true$.

Until new_row_added or all elements of S were processed

Until $\neg new\_row\_added$

Construct $M(S, W, T)$.

Return $M(S, W, T)$.

Figure 13.4: The learning procedure *LearnDFCA*

been added to W as a result of an incorrect consistency check will be itself processed in the same “For” loop.

- In order to check closedness, the procedure will search for $s \in S$ and $a \in A$ such that sa is dissimilar to any of the current rows t for which $\|t\| \leq \|sa\|$; similarly, the search is performed in increasing order of length of s . If such s and a are found, then sa is added to the observation table and the algorithm will check again its consistency.

Consider once again Table 13.2 (left hand side) as the current observation table. This will fail the consistency check for $i = 0$ and $k = 2$: $s_1 = \epsilon$, $s_2 = b$, $w = \epsilon$ and $\alpha = b$ satisfy $s_1 \sim_k s_2$, but $T(s_1\alpha w) \neq T(s_2\alpha w)$. Consequently, $\alpha w = b$ is added to W and so Table 13.2 (right hand side) is

the resulting observation table. This is both consistent and closed and so the DFA $M(S, W, T)$ is constructed.

The state set of DFA $M(S, W, T)$ is formed by taking all minimum, mutually dissimilar sequences from S , where the minimum is taken according to the quasi-lexicographical order on A^* [Ipa11]. For Table 13.2 (right hand side), this is $Q = \{\epsilon, a, b, aa\}$ (since $bb \sim aa$) and the corresponding DFA is as represented in Fig. 13.5 (in which final states are drawn in double line, whereas non-final states are drawn in single line; the initial state is q_0). The formal definition of $M(S, W, T)$ is given in [Ipa11], for simplicity this is not reproduced here. Further details regarding the L^ℓ algorithm, including proofs of correctness and termination and examples which illustrate its functioning, can also be found in [Ipa11].

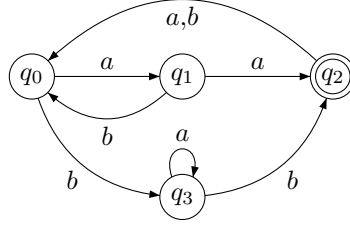


Figure 13.5: The DFA corresponding to Table 13.2 (right hand side)

13.4.2 Black-box testing for finite cover automata

Before proceeding, we briefly outline the W -method for bounded sequences, which we proposed in [Ipa10]. This is not central to our approach, but may be used for answering language queries, as discussed later.

Given a DFCA model $M = (A, Q, q_0, F, h)$ of a system and an upper bound ℓ , this method generates a set of sequences to check if the implementation under test, which is modeled by an unknown finite automaton I , behaves as defined by M for all sequences of length at most ℓ . In other words, if the languages accepted by M and I are L_M and L_I , respectively, the W -method will construct a finite set of sequences $X \subseteq A[\ell]$ such that $L_M \cap X = L_I \cap X$ implies $L_M \cap A[\ell] = L_I \cap A[\ell]$.

The implementation is a black box and so, naturally, I is not known; however, it is assumed that the maximum number of states of I can be estimated; the difference between this estimated maximum and the number of states of M is denoted by k (if the difference is negative then we take $k = 0$).

Naturally, one can always take X to be the set of all sequences of length up to ℓ ; however, the W -method produces a much smaller set, whose size

is polynomial in the number of states of M (but exponential in k). The construction of X is based on two sets: a *proper state cover* S and a *strong characterization set* W of M . S is called a proper state cover of M if it contains sequences of minimum length that reach all states of M , i.e. for every $q \in Q$ there exists $s \in S$ such that: $h(q_0, s) = q$ and $\|s\| \leq \|t\|$ for every $t \in A^*$ for which $h(q_0, t) = q$. W is called a strong characterization set if it contains sequences of minimum length that distinguish between any pair of states of the DFCA, i.e. for every $q_1, q_2 \in Q$, $q_1 \neq q_2$ there exists $w \in W$ such that: w distinguishes between q_1 and q_2 and $\|s\| \leq \|t\|$ for every $t \in A^*$ which distinguishes between q_1 and q_2 . Then, for an estimated value of k , the test set has the form $X_k = SA[k + 1]W \cap A[\ell]$.³

The W -method for bounded sequences [Ipa10], is a non-trivial generalization of the W -method for checking functional equivalence (also called the Vasilevski-Chow method).⁴

13.4.3 Model learning and test generation

We are now ready to present how we can apply the above theoretical harness. We first describe the Event-B modeling environment. Then we present our approach for incremental cover automata learning and test generation for Event-B models. The notion of refinement is also discussed at the end of the section.

Usually, in an Event-B model, all states which can be reached by feasible sequences of events are considered to be *final* states. Obviously, in this case only one non-final state is sufficient - a “sink” state which collects all infeasible paths. Although this is the situation we have encountered in all applications considered in this section, our approach is in no way restricted to this particular case. Furthermore, it would also be possible to differentiate between reachable and final configurations in an Event-B model by using a logical predicate on the global variables, i.e. a state is considered final only if the predicate holds.

Given an Event-B model, a *test case* can be defined as a sequence of events. This can be either positive, if it corresponds to an executable (feasible) path through the Event-B model, or negative, otherwise. The executability of a test case implies the existence of appropriate test data for the events, i.e. appropriate values for the local parameters that ensure that the

³The model M is assumed to be a minimal DFCA of $L_M \cap A[\ell]$ w.r.t ℓ (if not, it is minimized before the method is applied), so both a proper state cover and a strong characterization set exist.

⁴[Ipa10] gives the results for Mealy machines, whereas here we adapted them for finite state machine acceptors.

Procedure IterativeConstructionDFCA

Input: S_0, W_0

Set $S = S_0$ and $W = W_0$

Construct T for $(S \cup SA)W$

LearnDFCA

While the constructed automaton $M(S, W, T)$ is not correct **do**

 Provide a counterexample w

 Add w and all its prefixes to S

 Extend the observation table T to $(S \cup SA)W$

 LearnDFCA

Minimize S and W

Output: $M(S, W, T)$, (minimized) S and W , observation table
and the corresponding test sequences

Figure 13.6: The iterative procedure of constructing the DFCA

guard of the event is true. Finally, a *test suite* is by definition a collection of test cases.

Incremental model learning

We will apply now the cover automata learning method of Section 13.4.1 to the Event-B framework. The input elements for the procedure were a finite language U and a bound ℓ . For an Event-B model, U will be the set of all executable event sequences of length maximum ℓ . The alphabet of U is the set of events in the model, which we denote by A .

Given the above U and ℓ , our approach gradually constructs both (1) a DFCA for the Event-B model and (2) an associated test set. The test set will be constructed using information from the observation table (paths through the model) and the actual test data to drive the executable paths. In this subsection we discuss the model learning cycle and in the following two subsections the test suite creation.

The proposed procedure consists of a number of steps; at each step, a new DFCA and test set is produced. The outline of this procedure is depicted in Fig. 13.6. Unlike the original L^ℓ algorithm, the procedure does not start with empty S and W , but with some initial values S_0 and W_0 , which reflect the current knowledge about the DFCA model.

In case S_0 and W_0 are carefully chosen by a human with a good insight in

the model, the constructed $M(S, W, T)$ will be close to the correct DFCA and so the “while”-loop will be executed fewer times or not at all, saving in this way computational resources. At limit, when either S_0 or W_0 are correctly chosen from the outset, the constructed $M(S, W, T)$ will be correct and the “while”-loop will not be executed at all.

Otherwise, whenever the DFCA is found to be inaccurate, a counterexample (i.e. a sequence s with $\|s\| \leq \ell$ such that $s \in U$ but s is not accepted by the DFCA or vice versa) must be found and the observation table should be extended accordingly. Practical modalities for finding such a counterexample will be discussed later in this section.

Therefore, two main cases can be distinguished:

- **Case 1:** The procedure is executed for the first time. In this case, the initial sets S_0 and W_0 are based on an initial estimation of the states of the model. In the worst case (when no initial estimation is available), we take $S_0 = \{\epsilon\}$, $W_0 = \{\epsilon\} \cup A$. (As it emerged from our empirical evaluation (Section 13.4.5), in many cases states can be distinguished by singleton sequences, and so initially we consider W to contain all event names, i.e. A).
- **Case 2:** The procedure has already been applied at least once and, consequently, a DFCA model exists. Suppose that this model is not totally accurate and needs to be improved. This may happen for a number of reasons:
 - **Subcase 1:** the Event-B model has been modified or augmented due to changes in the requirements.
 - **Subcase 2:** the Event-B model has not been changed but the associated DFCA is deemed to be insufficient for testing purposes. In this case, the upper bound ℓ is increased according to the existing testing needs and the procedure is executed once more for the new value of ℓ .
 - **Subcase 3:** the existing Event-B model has been refined and extra detail has been added (using the Event-B refinement). This subcase will be discussed later in subsection 13.4.4.

In this (second) case, S_0 and W_0 are the values of S and W from the previous iteration. In fact, it is not necessary to reuse the entire sets S and W . It is sufficient to extract from them two minimal subsets $S_{min} \subseteq S$ and $W_{min} \subseteq W$, where S_{min} is the set of all minimum, mutually dissimilar sequences from S , and W_{min} is the set of minimum

sequences from W which distinguish between any two dissimilar sequences of S - this corresponds to the minimization step in Fig. 13.6.⁵ The construction of S_{min} and W_{min} is not computationally expensive; both these subsets are selected by simply scanning the observation table, so the complexity is linear in its size. Additionally, S_{min} is actually the state set of $M(S, W, T)$, so it is computed by the algorithm anyway. The observation table T (corresponding to the minimized S and W) is also partially/totally re-constructed in the next iteration as follows. In the first subcase, since the Event-B model has been changed, the value of T must be re-checked for all sequences in $(S \cup SA)W \cap A[\ell]$. In the second subcase, only the sequences in $(S \cup SA)W$ whose length is greater than the previous ℓ need to be re-processed.

Note that (Case 2 / Subcase 1) can also be applied even if the procedure has not been applied before but an automaton model of the system exists from other sources (e.g. has been developed during the design phase), but has become obsolete. In this case, S_{min} and W_{min} can also be derived from the existing model, as explained above, so the information contained in the existing model is reused in the construction of the new model,

Example. We illustrate the iterative process of constructing the DFCAs with a system for controlling the cars on a narrow bridge between an island and the mainland. This example is particularly relevant since it is used to introduce the main Event-B concepts in Abrial's textbook [Abr10].

The modeled system is equipped with two traffic lights with two colors: green and red. The traffic lights control the entrance to the bridge at both ends. Cars are not supposed to pass on a red traffic light, only on a green one. There are also some car sensors situated at both ends of the bridge which are used to detect the presence of a car entering or leaving the bridge. The system has two main additional constraints: the number of cars on the bridge and island is limited and the bridge is one-way.

We present here only the first two levels of refinement (see Fig. 13.7 and Fig. 13.8). The first model M_0 is very simple. The events ML_out and ML_in correspond to cars entering and leaving the island-bridge compound, respectively. The context contains a single constant d , which is a natural number denoting the maximum number of cars allowed to be on the island-bridge compound at the same time. The single variable n of the machine M_0 denotes the actual number of cars.

In the first refinement, the machine M_1 introduces the bridge. The events

⁵Intuitively, this is because S_{min} and W_{min} still remain a proper state cover and a strong characterization set of $M(S, W, T)$, so the language equivalence (modulo the upper bound ℓ) against any other automaton with the same number of states is ensured.

ML_out and ML_in correspond now to cars leaving the mainland and entering the bridge or leaving the bridge and entering the mainland, respectively. In addition, the events IL_in and IL_out correspond to cars entering and leaving the island, respectively. The variable n is now replaced by three variables: a (the number of cars on the bridge and going to the island), b (the number of cars on the island) and c (the number of cars on the bridge and going to the mainland).

Finally, the second refinement introduces the two traffic lights, named ml_tl and il_tl . The model M_2 has two new events to turn the value of the traffic lights color to green when they are red: ML_tl_green and IL_tl_green . In order to make the colors change in a more disciplined way, two more variables ml_pass and il_pass are introduced: $ml\_pass = TRUE$ signifies that at least one car has passed the bridge going to the island since the mainland traffic light last turned green; similarly for $il\_pass = TRUE$.

First, we start the learning process with $\ell = 3$ (*Case 1*); for each of the three models, the procedure is executed with initial values $S_0 = \{\epsilon\}$ and W_0 equal to the corresponding input alphabet plus the empty sequence; the resulted DFCA's (plotted using our plug-in) are presented on the right hand side of Fig. 13.7 (for simplicity, the sink states are not shown).

Suppose now that we want to improve the DFCA for M_2 by increasing the upper bound ℓ (*Case 2 / Subcase 2*). For $\ell = 6$, we obtain the DFCA in Fig. 13.9(a), which has more states and transitions (and covers events like ML_in that were not covered for $\ell = 3$).

In order to illustrate the iterative DFCA construction (cf. Fig. 13.6), we provide a counterexample path for the current DFCA associated to M_2 . For instance, the following sequence of length 5:

$$w = ML\_tl\_green, ML\_out1, ML\_out2, IL\_in, IL\_in$$

is feasible in M_2 , but it is not accepted by the DFCA in Fig. 13.9(a). The new DFCA taking into account the counterexample w (see the while-loop of Fig. 13.6) is presented in Fig. 13.9(b). It can be observed that the path $q_0 \rightarrow q_2 \rightarrow q_3 \rightarrow q_6 \rightarrow q_4 \rightarrow q_8$ in the new DFCA corresponds to the path w in M_2 .

Naturally, finding a counterexample is the most problematic part of our approach and a model checker such as ProB can be of great assistance. There are several possibilities to do this:

- interactively, using the experience of the human testers that have a good understanding of the model: Testers can use the simulation and animation capabilities of ProB to discover counterexamples, that are fed to the learning algorithm. Moreover, high-priority scenarios that

```

Machine  $M_0$ :
  Variables:  $n$ 
  Event INITIALISATION  $\hat{=}$  begin  $n := 0$  end
  Event  $ML\_out$   $\hat{=}$  when  $n < d$  then  $n := n + 1$  end
  Event  $ML\_in$   $\hat{=}$  when  $n > 0$  then  $n := n - 1$  end

```

```

Machine  $M_1$  refines  $M_0$  :
  Variables:  $a, b, c$ 
  Event INITIALISATION  $\hat{=}$  begin  $a, b, c := 0, 0, 0$  end
  Event  $ML\_out$  refines  $ML\_out$   $\hat{=}$  when  $a + b < d \wedge c = 0$  then  $a := a + 1$  end
  Event  $ML\_in$  refines  $ML\_in$   $\hat{=}$  when  $c > 0$  then  $c := c - 1$  end
  Event  $IL\_in$   $\hat{=}$  when  $a > 0$  then  $a, b := a - 1, b + 1$  end
  Event  $IL\_out$   $\hat{=}$  when  $0 < b \wedge a = 0$  then  $b, c := b - 1, c + 1$  end

```

```

Machine  $M_2$  refines  $M_1$  :
  Variables:  $a, b, c, ml\_tl, il\_tl, il\_pass, ml\_pass$ 
  Event INITIALISATION  $\hat{=}$  begin  $(a, b, c := 0, 0, 0),$ 
                                 $(ml\_tl, il\_tl := red, red),$ 
                                 $(il\_pass, ml\_pass := 1, 1)$ 
                                end
  Event  $ML\_out1$  refines  $ML\_out$   $\hat{=}$  when  $ml\_tl = green \wedge a + b + 1 < d$ 
                                then  $a, ml\_pas := a + 1, 1$  end
  Event  $ML\_out2$  refines  $ML\_out$   $\hat{=}$  when  $ml\_tl = green \wedge a + b + 1 = d$ 
                                then  $(a, ml\_pas := a + 1, 1), ml\_tl := red$  end
  Event  $IL\_out1$  refines  $IL\_out$   $\hat{=}$  when  $il\_tl = green \wedge b > 1$ 
                                then  $(b, c := b - 1, c + 1), il\_pas := 1$  end
  Event  $IL\_out2$  refines  $IL\_out$   $\hat{=}$  when  $il\_tl = green \wedge b = 1$ 
                                then  $(b, c := b - 1, c + 1), (il\_tl, il\_pas := red, 1)$ 
                                end
  Event  $ML\_tl\_green$   $\hat{=}$  when  $ml\_tl = red \wedge a + b < d \wedge c = 0 \wedge il\_pass = 1$ 
                                then  $(ml\_tl, il\_tl := green, red), ml\_pass := 0$  end
  Event  $IL\_tl\_green$   $\hat{=}$  when  $il\_tl = red \wedge 0 < b \wedge a = 0 \wedge ml\_pass = 1$ 
                                then  $(ml\_tl, il\_tl := red, green), il\_pass := 0$  end
  Event  $IL\_in$  refines  $IL\_in$   $\hat{=}$  when  $a > 0$  then  $a, b := a - 1, b + 1$  end
  Event  $ML\_in$  refines  $ML\_in$   $\hat{=}$  when  $c > 0$  then  $c := c - 1$  end

```

Figure 13.7: The first two refinements of the "Cars on the bridge" example (from Abrial [Abr10])

the testers deem as important can be introduced into the learning loop and the associated tests will be covered by the DFCA.

- by testing language equivalence, using the W -method for bounded sequences outlined in section 13.4.2: Recall that, given a DFCA model

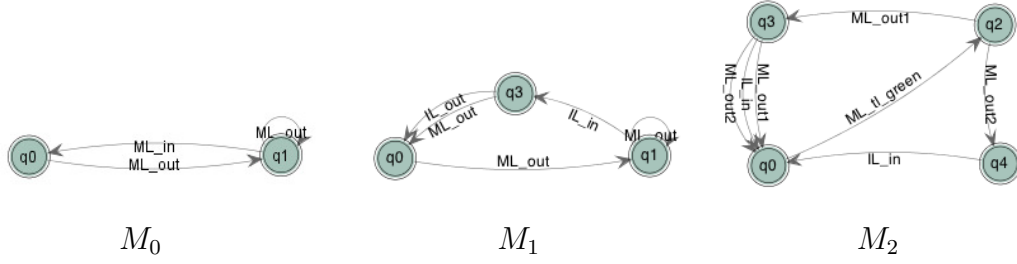


Figure 13.8: DFCA for CarsOnBridge given $\ell = 3$

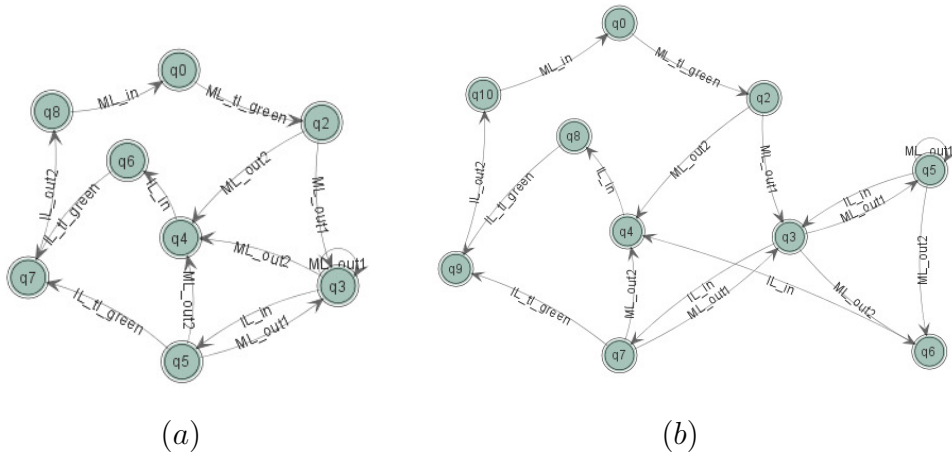


Figure 13.9: DFCA for (a) M_2 and $\ell = 6$; and (b) the DFCA improvement after providing a counterexample.

M of a system and an upper bound ℓ , this method generates a test set to check if the implementation under test, modeled by an unknown automaton I , behaves as defined by M for all sequences of length at most ℓ . The test set has the form $X_k = SA[k+1]W \cap A[\ell]$, where S and W are a proper state cover and a strong characterization set of M , respectively, and k is the difference between the estimated maximum number of states of I and the number of states of M . In our case, the model M corresponds to the current DFCA $M(S, W, T)$ and the implementation under test to the Event-B model (more precisely, an approximation of the Event-B model, which contains all set of executable paths of length up to ℓ). Now, the sets S and W in the observation table satisfy the definitions of a proper state cover and a strong characterization set of $M(S, W, T)$, respectively. Thus, for $k = 0$, the test set X_0 is actually the set of sequences in the observation table, so, if the Event-B model is known to have no more states than the current DFCA, this step is already completed. Otherwise, testing the behavioral equivalence between the current DFCA and the Event-B model corresponds to gradually increasing k until a counterexample is found (a test case produces a different result on the Event-B model compared to the DFCA) or we are satisfied that the DFCA is correct. Note that the size of the test set is exponential in k and so using the W -method for a large k may be expensive.

- by encoding the language equivalence problem into the ProB model checker: For instance the complement of the DFCA is encoded into a CSP process P and ProB will try to run the Event-B machine and P in parallel to find a path that is accepted by Event-B but not by the DFCA. Note, however, that this procedure might be computationally expensive.

From the three options above, in the current version of our implementation we only consider the first one, in which the counterexample is manually provided (which is in fact in the spirit of the original Angluin's algorithm). This also fits well with common practice, in which human knowledge is used to guide model and test design.

Test data generation

In order to decide whether a given sequence s , $\|s\| \leq \ell$, is accepted or not by the DFCA (i.e., $s \in L^\ell$ or not), the procedure needs to check if s is a feasible path through the Event-B model. This is achieved by effectively constructing (or attempting to construct) test data to drive the given path.

If the appropriate test data has been found, then $s \in L^\ell$; otherwise, the path is declared infeasible⁶ and so $s \notin L^\ell$. Therefore, deciding whether $s \in L^\ell$ or not reduces to finding test data to execute the corresponding path of the Event-B model.

Then, all it remains to specify are the method(s) used to find test data to execute a given path of an Event-B model. So far two such approaches have been proposed and implemented. The first used symbolic execution and reduces this problem to solving a set of constraints [LB08]. The second reduces the problem to an optimization problem, which is then solved using search-based techniques (genetic algorithms) [DSI⁺11]. Note that the test data generation problem may be complex even for one path, when the guards are complex and the test data domain are large. In particular, the set-theoretic nature of Event-B increases the search space because free set variables v that are subsets of a given carrier set V (i.e. $v \subseteq V$) can take exponentially many values, $2^{\text{card}(V)}$. Consequently, most of the time taken by the execution of the procedure is spent on generating the actual test data.

13.4.4 Test suite construction

When the process of constructing the DFCA is completed, a test suite for the Event-B model has also been obtained; this is precisely the set of sequences in the observation table returned by the procedure, $X_0 = (S \cup SA)W \cap A[\ell]$ (note that the procedure returns the minimized S and W and so these are the used in the definition of X_0). The test sequences can be classified into *positive* (for which $T(x) = 1$, which correspond to feasible paths in the Event-B model) and *negative* (for which $T(x) = 0$). Naturally, test data can only be generated for feasible paths and, as explained earlier, test data generation is implicitly included in the DFCA construction procedure. Negative test sequences are also useful for testing the system implementation since they describe erroneous scenarios, which the system cannot perform in normal functioning.

Following [Ipa10], the constructed set will constitute a conformance test suite for the Event-B model modulo the bound ℓ (the ℓ -bounded behavior of the model). Such a test is more powerful than a set of tests based on state or transition coverage criteria since it covers all states and all transitions of the equivalent automaton and also checks each state and the initial and destination states of each transition. Conformance testing is especially relevant

⁶Note that here *infeasible* means only that our tools could not find test data within reasonable time (e.g. 20 seconds for one path) and we stop searching, whereas in reality there might exist such test data. However, since we are working with approximated models, this incompleteness aspect is not very important.

in the embedded systems domain.

Increasing ℓ , longer and more complex tests are generated. However, very complex or long test sequences are usually not the norm, so having the ability to tune the length of the test case using ℓ is an advantage of our approach. Another advantage is the fact that the method is interactive, so the tester can use its intuition to provide relevant sequences to the algorithm to learn and thus more directly influence the result of the test suite. This is in contrast to purely automatic test generation techniques that are driven by coverage criteria, where the produced tests may not be intuitive or may not cover existing standard testing scenarios in the domain. Regarding coverage criteria, if a very specific coverage criteria is sought (cf. [GR01]), our method can accommodate this to some extent in that the training set of sequences for the learning algorithm can be chosen according to the desired coverage. Moreover, if a simpler coverage criteria like event coverage is desired, the obtained test suite can be reduced by choosing a smaller subset that satisfies the requirement.

Relation to Event-B refinement

Very often, model design is an iterative process, in which, at each step, the existing (more abstract) model is replaced by a more concretized model through *refinement*. The incremental approach of L^ℓ allows us to reuse the learned model and test suite of the abstract model to the next more concretized model (Case 2, Subcase 3 from subsection 13.4.3), as explained below. Again, the approach is presented in the context of Event-B, but the basic ideas can be extended to other languages which provide model refinement as a way to handle complexity.

Suppose we have a refinement from AM (abstract model) to CM (concrete model). In a refinement step, new events can be introduced and the existing events can be made more concrete. Let A and A' denote the sets of events of AM and CM , respectively, and let $E \subseteq A'$ be the new events introduced in CM that do not refine any abstract event. Every abstract event $a \in A$ from AM will correspond to a set (containing one or many concrete events) from CM .⁷ Let us denote this set $ref(a)$, $ref(a) \subseteq A' \setminus E$. Also, let $ref(A) = \{ref(a) \mid a \in A\}$. Suppose S_{min} and W_{min} are the (minimized) sets produced by the application of our procedure on the abstract model AM . As these sets contain sequences of abstract events, they need to be transformed before they can be reused in the construction of the automaton corresponding to the concrete model CM . On the other hand, it can be shown that only

⁷In general the correspondence may be many-to-many but in the vast majority of the models we have encountered a one-to-many correspondence is sufficient.

Transformation of S

Input: S_{min} and the restriction of T to S_{min}
 $maps(\epsilon) = \epsilon$
 $Y = S_{min} \setminus \{\epsilon\}$
While $Y \neq \emptyset$ **do**
 Select $x = sa$, $s \in A^*$, $a \in A$, a seq. in Y of minimum length
 $s' = maps(s)$
 If $T(x) = 1$ **then**
 $found = find\_next(s', a, t)$
 If $found$ **then**
 $maps = maps \oplus (x, s't)$
 $Y = Y \setminus \{x\}$
 Else
 If $s = \epsilon$ **then**
 Return failure
 Else
 $Y = Y \cup (dom(maps) \cap \{s\}A[1])$
 $dom(maps) = dom(maps) \setminus \{s\}A[1]$
 Else
 $maps = maps \oplus (x, s'a')$ for some $a' \in ref(a)$
 $Y = Y \setminus \{x\}$
Return $S_{min}^R = pref(Im(maps))$

Figure 13.10: The transformation of S in the case of refinement

one of the two sets (S_{min} or W_{min}) is sufficient to correctly determine the corresponding DFCA model (the other set is reconstructed by the algorithm). Furthermore, the set of all feasible paths of an Event-B model is closed under prefixing and so, naturally, a path from AM is transformed into a path from CM by gradually transforming its prefixes. Such a transformation is natural in the case of S , which is prefix-closed, but is problematic for W , which must be suffix-closed. For these reasons we choose to only transform the set S_{min} . For W , we will use the same type of heuristic as for the case in which the DFCA construction procedure is executed for the first time: W is initialized with the set A' of all events of CM (along with the empty sequence).

The transformation of S_{min} is given in pseudocode in Fig. 13.10; $maps$ denotes the mapping between each sequence in S_{min} and the corresponding sequence in the concrete model. Ultimately, the algorithm will return the

prefix-closure of the image of map_S , $pref(Im(map_S))$. Y denotes the set of sequences from S_{min} that remain to be processed. Sequences are processed in increasing order of their length, so, at any time, a (the) sequence of minimum length from Y is selected. As S_{min} is prefix-closed, $map_S(x)$ is obtained by extending the transformation of its longest prefix s ($x = sa$, $s \in A^*$, $a \in A$). Two main cases can be distinguished.

- If x is a feasible path of AM , then $map_S(x)$ must also be a feasible path of CM . This is obtained by extending $s' = map_S(s)$ with a sequence $t = e_1 \dots e_j a'$, where $e_1, \dots, e_j \in E$, $j \leq k$ (k is a predefined upper bound), and $a' \in ref(a)$. The function $find\_next(s', a, t)$ searches for such a t in increasing order of j ; if found, the function will return *TRUE*, otherwise *FALSE*. $find\_next$ may be called several times with the same input parameters s and a' during the execution of the algorithm. Each time, it continues the search from where it left off, so each time a different solution is produced. If $find\_next$ cannot find a (new) solution, the algorithm backtracks: it removes $s \neq \epsilon$ and all sequences which extend s from the domain of map_S and adds them to Y . Consequently, the algorithm will resume by processing s . If $s = \epsilon$, the algorithm cannot backtrack any further; in this case, it stops and reports failure.
- If x is not feasible in AM then s' can be extended with any $a' \in ref(a)$.⁸ Note that S_{min} contains only one such sequence since the corresponding automaton will have only one non-final (“sink”) state.

In refinement, an event a from AM is replaced by (one or more) events $ref(a)$ in CM , describing the system reactions in different circumstances. Furthermore, the applications of the extra events may also condition the event operation and so each application of a in AM is replaced by some sequence $e_1 \dots e_j a'$ in CM , with $e_1, \dots, e_j \in E$ and $a' \in ref(a)$.

The new events from E cannot be indefinitely enabled [Abr10] and, furthermore, in practice it is reasonable to expect that an upper bound k on the number of times they can be applied in the absence of an event from $A' \setminus E$ can be established, and so $j \leq k$; the upper bound k is then used in the definition of the $find\_next$ function presented above. Thus, any feasible path $a_1 \dots a_n$ in the abstract model can be mapped (not necessarily in a unique fashion) onto a feasible path $u_1 a'_1 u_2 a'_2 \dots u_n a'_n$ in the concrete model, with $a'_i \in ref(a_i)$ and $u_i \in E[k]$, $1 \leq i \leq n$. This ensures that the transformation procedure will end successfully, so every sequence in S_{min} is refined

⁸Since the concrete guard is not weaker than the abstract one [Abr10], a non-feasible path in AM can only give rise to non-feasible paths in CM .

appropriately. Finally, we need to ensure that S_{min}^R is prefix-closed and so we take the prefix-closure of the refined sequences. Note that the transformation procedure given in Fig. 13.10 is only guaranteed to terminate successfully if every feasible path in AM has a corresponding feasible path in CM . This condition is in the spirit of refinement and is satisfied by the vast majority of the applications we have encountered. However, the algorithm can be easily extended so that it terminates successfully even when not all sequences in S_{min}^R can be refined - for simplicity and due to its reduced practical value, this idea is not pursued here.

Once the set S_{min} has been transformed, the DFCA construction procedure can be executed for the concrete model CM with initial values $S_0 = S_{min}^R$ and $W_0 = \{\epsilon\} \cup A'$. The upper bound ℓ^R used for the concrete model CM also needs to be established. This will be set by the user, but, naturally, it will be greater than or equal to the length of the longest sequence in S_{min}^R plus one.

Naturally, the DFCA construction procedure can always be applied directly on the concrete model, but the strategy presented here, which reuses the information from the abstract model in the construction of the concrete model and of its associated test set, presents some key advantages:

- There is a significant number of cases (60 percent of the models considered in our experiments, see Table 13.4) for which the “reuse” strategy produces richer DFCA than those produced “from scratch” (i.e. by the direct application of the procedure on the concrete model); on the other hand, in no case in our experiments the “from scratch” strategy produced a better model. A richer model (with a larger state space) represents a better approximation of the real system and is essential for the effectiveness of the resulting test cases. In order to obtain a more precise model, appropriate counterexamples must be supplied (and the DFCA construction procedure be run at least once more); this may add significant complexity to the test generation process.
- Even if the “reuse” strategy does not directly produce improved DFCA models, it is still preferable as it offers a way of incorporating the human knowledge into the model at the appropriate level of abstraction. Let us consider two Event-B models, AM (abstract) and CM (concrete), as above. It is likely that the first DFCA for AM is not satisfactory, so a richer automaton is obtained by supplying the construction procedure with appropriate counterexamples. In the “reuse” strategy, these counterexamples are propagated to the next level - they are implicitly included in the DFCA for CM . On the other hand, when the DFCA for CM is produced from scratch, all counterexamples must be produced at

this level. Naturally, abstract models are simpler than concrete models, so finding counterexamples (by human intervention or automatically) for the abstract model is simpler.

13.4.5 Experimental results

In this subsection we provide the results of our experimentation on a comprehensive benchmark of Event-B models. The implementation of our algorithms was done in Java as an Eclipse plugin⁹ to Rodin platform (in its latest version 2.3). The membership queries were implemented using the constraint-solving functionality of ProB and a timeout of 20 seconds per query was imposed. We run ProB with fixed internal parameters, although fine-tuning of ProB parameters may improve results in certain cases. The experiments were conducted on a Windows 7 Professional 64-bit machine with an Intel Core i7 2.80GHz (8 CPUs) processor and 12 GB of RAM.

We used a broad range of models for experimentation, including systems from the embedded systems, transportation and aerospace industries as well as academic and pedagogical Event-B models used in the literature. All the chosen 10 Event-B models are publicly available in the DEPLOY model repository¹⁰. Table 13.3 presents the models together with their complexity, i.e. number of refinements, number of events for each refinement and number of variables for each refinement. For instance, the third model is our running example "CarsOnBridge" (see Fig. 13.7). It has 3 refinements (M0/M1/M2/M3); each level (including the initial machine M0) has 3, 5, 9, and 17 events, respectively; and 1, 3, 7, and 18 variables¹¹, respectively. Note also that many of the models are rather complex, e.g. TrainCtrller exhibits 8 levels of refinements and the last level has 43 events and 35 variables.

Table 13.4 presents the results of our LearnDFCA procedure for the 10 models. For each model we considered for exemplification several machines (at different refinement levels) and different values of the bound ℓ . For each combination, we provide the number of states of the learned DFCA together with number of associated conformance tests and the number of iterations needed to generate the DFCA (i.e. how many times the outmost loop in Fig. 13.4 was executed). These three dimensions are given for two strategies: first one, named "from scratch" where information from the previous refinements

⁹Installation instructions and screenshots can be found at: http://wiki.event-b.org/index.php/MBT_plugin

¹⁰<http://deploy-eprints.ecs.soton.ac.uk>

¹¹Variables might have an integer type, but also more complex types like sets, relations or partial functions, which increase the complexity of the algorithms, especially the membership queries.

is not used, and second, where an information reuse is taken into account as proposed in previous Section 13.4.4. As hoped, the "reuse" approach generated richer DFCA's in a smaller number of iterations.

We note that the number of produced tests may seem high, but this is because conformance testing is a strong form of testing heavily exercising the system. However, more compact test suites can be obtained according to weaker coverage criteria. For instance, test suites for state and transition coverage are readily available from the learning procedure, from the sets S and $S \cup SA$, respectively. Moreover, we have also implemented different test suite optimization algorithms that produce significantly smaller test suites.

A practical heuristic: In our experiments we have found that, in most cases, the states of the resulting automata can be distinguished by using only singleton sequences. Since these are already contained in the initial W , in these case the consistency checks performed by the DFCA are already successful and produce no effect on the observation table (the set W is not expanded). Even when there are states which can only be distinguished by sequences longer than 1, generally W is expanded much less frequently than S and so consistency checks would fail much more rarely than closedness checks. This observation led to the following heuristic: the procedure is applied first without consistency checks (at this step only S is enlarged as a consequence of the failed closedness checks) and then, once more, in which both consistency and closedness checks are performed. The heuristic should considerably reduce the overall number of consistency checks. Since consistency checks are time consuming operations (each such operation may require a large number of membership queries), the heuristic should also reduce the execution time. We have implemented both variants of the procedure (the original version and one which uses the aforementioned heuristic). The experimental results shown in Table 13.5 indeed show that the heuristic produced improved results in the majority of cases, but more significantly, that the improvements are significant (over 50%) for large execution times (over 1,000 s).

Benchmark: We provide a short description of the Event-B models in Table 13.3 including pointers where they can be retrieved:

1. *A2A* - <http://deploy-eprints.ecs.soton.ac.uk/129/> - Business domain: A model of the Order and Supply Chain A2A Communication using a pattern approach from ETH Zurich.
2. *BepiColombo* - <http://deploy-eprints.ecs.soton.ac.uk/72/> - Aerospace domain: A model of two communication modules in the embedded software on a space craft (BepiColombo mission). The model was con-

Table 13.3: The complexity dimensions of the ten subjects (number of refinements, number of events and number of variables)

Subject	# of refin's	# of events per refin.	# of variables per refin.
A2A	12	4/5/5/7/9/9/12/14/15/ 16/16/17/17	2/2/4/8/8/10/11/12/13/ 15/16/18/17
BepiColombo	3	6/11/13/17	6/10/12/18
CarsOnBridge	3	3/5/9/17	1/3/7/18
CircArbiter	4	8/8/8/8/8	7/9/11/10/10
Choreography	1	7/13	7/17
MobileAgent	5	5/7/7/8/8/8	3/5/5/7/7/7
PressCtrller	7	5/13/17/17/21/21/21/29	2/6/8/8/10/10/11/15
ResponseCoP	3	5/14/17/17	3/4/5/6
SSFPilot	3	14/20/23/41	5/7/8/14
TrainCtrller	8	8/10/15/20/20/27/38/43/43	5/6/9/14/14/15/27/33/35

structed by researchers in Southampton [FRB11] based on the feedback from SSF.

3. *CarsOnBridge* - <http://deploy-eprints.ecs.soton.ac.uk/112/> - Pedagogical example: Described in Section 13.4.3 and [Abr10, Chapter 2].
4. *CircArbiter* - <http://deploy-eprints.ecs.soton.ac.uk/117/> - Pedagogical example: Event-B model of the synchronous electronic circuits, see also [Abr10, Ch. 8].
5. *Choreography* - see [WKR⁺09] - Business domain: A model of service choreography for enterprise component communication.
6. *MobileAgent* - <http://deploy-eprints.ecs.soton.ac.uk/120/> - Distributed systems domain: A model for distributed computing communication: a routing algorithm for sending messages to a mobile phone, see also [Abr10, Chapter 12].
7. *PressCtrller* - <http://deploy-eprints.ecs.soton.ac.uk/113/> - Embedded control domain: A model of a mechanical press controller adapted from a real system at INRST (Institut National de la Recherche sur la Sécurité du Travail), see also [Abr10, Chapter 3].
8. *ResponseCoP* - <http://deploy-eprints.ecs.soton.ac.uk/301/> - Information flow domain: A model for analysis of information flow policies for Dynamic Virtual Organisations (DVOs), commonly referred to as the Bronze/Silver/Gold structure that frequently arises in multi-agency response to emergencies.
9. *SSFPilot* - <http://deploy-eprints.ecs.soton.ac.uk/58/> - Aerospace domain: A model of a pilot for a complex on-board satellite mode-rich system: Attitude and Orbit Control System (AOCS).
10. *TrainCtrller* - <http://deploy-eprints.ecs.soton.ac.uk/316/> - Automotive Domain: The model specifies a controller that detects the driving mode wished by the train driver. A large number of requirements are taken into account, therefore a large number of variables and events are needed.

13.4.6 Conclusions

In this section, we presented a novel approach of using model learning for testing purposes and its application to the Event-B method. This is based on

sound theoretical automata theory foundations and has an incremental and interactive nature that makes it fit the testing practice requirements. The prototype implementation showed that the method works well for realistic models, of medium or even fairly large size. As future plans, we want to further investigate the scalability of our approach and implementation on even larger models. Moreover, we plan to implement the language equivalence query as a means for interactively providing a counterexample (cf. end of Subsection 13.4.3), which is done manually in the current implementation. We will also implement different optimizations on our prototype, especially on the membership queries which constitute the most expensive part of the procedure. For instance, we can compute batches of the membership queries in parallel (on a multi-core/multi-processor architecture) or we can evaluate more sophisticated reductions, e.g. partial-order reductions by exploiting the independence of different events. Last but not least, we plan to extend the model learning and test generation to decomposed systems [Abr10, FRB11]; this is important for industry, where the complexity of the large specifications is tackled not only by successive refinements but also by model decompositions.

Table 13.4: Results for "From scratch" and "Reuse" strategies (number of states, number of test cases and number of iterations)

			"From scratch" (i.e. no reuse)			"Reuse"		
Subject	M	ℓ	# of states	# of tests	# of iter.	# of states	# of tests	# of iter.
A2A	M2	11	5	79	5	5	79	2
	M3	11	8	285	8	8	285	5
	M4	11	8	445	8	8	445	2
	M6	11	14	1518	14	14	1518	7
BepiColombo	M2	9	47	4798	47	51	5702	11
	M3	12	126	24202	126	131	26896	65
	M2	11	48	5098	48	53	6249	13
	M3	15	131	26958	131	145	34115	75
CarsOnBridge	M1	13	5	79	5	5	79	4
	M2	13	9	429	9	9	429	4
	M3	13	38	7407	38	54	11402	39
CircArbiter	M4	12	5	141	5	7	293	2
Choreography	M0	13	5	120	5	5	120	5
	M1	13	10	957	10	10	957	5
MobileAgent	M2	12	13	445	13	17	777	2
	M3	12	25	1162	25	33	1964	18
	M4	12	25	1162	25	33	1964	2
PressCtrller	M0	8	5	81	5	5	81	5
	M1	8	43	4857	43	46	5628	41
	M2	8	127	22017	127	136	26385	92
ResponseCoP	M0	3	4	41	4	4	41	4
	M1	8	37	3230	37	43	5091	38
	M0	4	5	51	5	5	51	5
	M1	9	49	5767	49	65	9539	58
SSFPilot	M0	10	54	7541	54	54	7541	54
	M1	10	96	20430	96	96	20430	95
	M1	11	107	25370	107	107	25370	107
TrainCtrller	M5	13	12	3117	12	12	3117	2
	M6	13	12	4437	12	12	4437	2
	M7	13	18	9054	18	18	9054	15
	M8	13	20	11203	20	20	11203	17

Table 13.5: Comparison of execution times for the original and the proposed heuristic algorithm (below, "time": execution time of the original learning algorithm, "time_h": execution time of the heuristic algorithm in Section 13.4.5, "time_tr": execution time of the transformation algorithm in Fig. 13.10)

		"From scratch" (i.e. no reuse)			"Reuse"			
Subject	M/ ℓ	time	time_h	Red.%	time	time_h	Red.%	time_tr
A2A	M2/11	0.06	0.06	0.00	0.05	0.05	-5.36	0.07
	M3/11	0.38	0.52	-35.60	0.50	0.50	-0.20	0.17
	M4/11	0.90	0.77	14.16	0.86	0.87	-0.93	0.25
	M6/11	1.07	1.07	0.00	0.01	0.01	-10.00	0.32
BepiColombo	M2/9	107.30	71.87	33.01	70.87	63.06	11.01	3.39
	M3/12	9708.96	3543.01	63.51	9085.45	3389.88	62.70	26.55
	M2/11	121.08	81.47	32.72	107.75	96.50	10.44	3.63
	M3/15	12273.73	4402.01	64.13	17620.10	7968.07	54.78	30.43
CarsOnBridge	M1/13	0.07	0.08	-4.17	0.08	0.08	0.00	0.07
	M2/13	0.90	0.91	-1.11	0.67	0.69	-3.58	0.29
	M3/13	140.79	81.26	42.28	537.30	299.60	44.24	5.36
CircArbiter	M4/12	0.78	0.51	34.57	1.73	0.91	47.69	0.80
Choreography	M0/13	0.45	0.47	-3.54	0.62	0.62	0.00	0.00
	M1/13	10.92	10.45	4.29	10.25	9.29	9.30	2.73
MobileAgent	M2/12	6.76	1.72	74.54	4.17	2.24	46.32	1.21
	M3/12	42.09	9.03	78.56	43.78	16.56	62.17	2.05
PressCtrller	M0/8	0.04	0.04	-15.79	0.03	0.06	-93.75	0.00
	M1/8	48.44	31.82	34.30	80.34	46.71	41.86	0.42
	M2/8	5658.31	2545.58	55.01	7944.75	3277.45	58.75	4.71
ResponseCoP	M0/3	0.06	0.061	-3.39	0.01	0.07	-6.56	0.00
	M1/8	37.33	28.57	23.64	161.96	119.40	26.28	2.62
	M0/4	0.09	0.10	-11.63	0.05	0.06	-11.76	0.00
	M1/9	125.06	79.02	36.81	775.88	555.16	28.45	2.66
SSFPilot	M0/10	186.71	116.87	37.40	186.71	112.61	39.69	0.00
	M1/10	5463.69	2371.34	56.60	5463.69	2596.46	52.48	0.71
	M1/11	8435.40	3652.08	56.71	9505.11	3945.82	58.49	0.58
TrainCtrller	M5/13	31.57	33.83	-7.17	29.73	31.54	-6.07	1.38
	M6/13	99.35	110.46	-11.18	100.37	104.75	-4.37	6.76
	M7/13	394.86	451.11	-14.24	389.24	440.22	-13.10	6.78
	M8/13	531.50	536.62	-0.96	53.00	526.55	0.65	6.78

13.5 Test suite reduction

Multi-objective test suite reductions Large test suites generated by automatic test generators usually need to be optimised according to different criteria. In the following, we follow a more general approach of defining the reduction criteria as multi-objective test suite optimization problems. They are solved using two modern Multi-Objective Evolutionary Algorithms, namely: NSGA-II [DAPM00] and SPEA-2 [ZLT01]. The experiments have been conducted using five test suites generated from two industrially-inspired Event-B models.

The rest of this section is structured as follows. We introduce the test suite optimization problem for Event-B models, then we mathematically define the six different test suite optimizations, and we finally describe the experiment setup and results.

The presented results are based on [Din11] and have used publicly available Event-B models rather than the SAP internal models.

13.5.1 Multi-Objective Test Suite Optimization

Given an Event-B machine M with $E = \{e_1, e_2, \dots, e_m\}$ the set of its events, a test case can be defined as a sequence of events in E that can be executed in the machine M (an execution path). Each test case begins with a special event called *INITIALISATION* which serves to initialize the global variables of the machine before starting the execution of a test case. A test suite is by definition a collection of test cases.

We introduce the multi-objective test suite minimization problem. We adopt here the definitions from [YH10]. Generally, a multi-objective optimization problem can be defined as to find a vector of decision variables x , which optimizes a vector of M objective functions $f_i(x)$, $1 \leq i \leq M$. The objective functions are the mathematical formulations of the optimization criteria. Usually, these functions are conflicting, which means that improvements with respect to one function can only be achieved when impairing the solution quality with respect to another objective function. Solutions that can not be improved with respect to any functions without impairing another one are called *Pareto-optimal solutions*.

Formally, let us assume that, without loss of generality, the goal is to minimize the functions $f_i(x)$, $1 \leq i \leq M$. A decision vector x is said to *dominate* a decision vector y (we write $x \succ y$) if and only if the following property is satisfied by their objective vectors:

$$f_i(x) \leq f_i(y), \forall i \in \{1, 2, \dots, M\} \text{ and } \exists i_0 \in \{1, 2, \dots, M\}, f_{i_0}(x) < f_{i_0}(y).$$

The dominance relations states that a solution x is preferable to another solution y if x is at least as good as y in all objectives *and* better with respect to at least one objective. The *Pareto-optimal set* is the set of all decision vectors that are not dominated by any other decision vectors. The corresponding objective vectors are said to form *Pareto frontier*. Therefore, the multi-objective optimization problem can be defined in the following manner:

Given: a vector of decision variables, x , and a set of objective functions, $f_i(x)$, $1 \leq i \leq M$,

Problem: minimize $\{f_1(x), f_2(x), \dots, f_M(x)\}$ by finding the Pareto-optimal set over the feasible set of solutions.

With respect to multi-criteria test suite optimization, the objective functions f_i are the mathematical descriptions of the testing criteria that must be satisfied to provide desired adequate testing of the model. In real industrial testing problems, there exist multiple test criteria, because a single ideal criterion is simply impossible to be achieved. For example, a frequently optimization problem is to produce a minimal test suite which achieves maximal coverage of the model entities with a minimal execution cost. Therefore, this is a *bi-objective minimization* test suite problem.

Formally, multi-objective test suite optimization problem can be defined in the following manner [YH10]:

Multi-Objective Test Suite Optimization.

Given: a test suite TS , a vector of M objective functions f_i , $1 \leq i \leq M$

Problem: to produce a subset $T \subset TS$, such that T is a Pareto-optimal set with respect to the set of the above objective functions.

In the following, we instantiate this general multi-objective test suite optimization problem with respect to our Event-B models.

We assume an Event-B machine M for which we have generated a test suite TS . Of course, TS satisfies a set of test requirements which are expressed as a level of coverage of the model. For the moment, we only consider that the test suite TS achieves the following simple coverage criterion:

Event Coverage Criterion: A test suite $TS = \{t_1, \dots, t_m\}$ of m test cases for an Event-B model M is said to achieve *event coverage criterion* if and only if for each event e of the model M there exists a test case $t_i \in TS$ which covers e .

Having the above criterion in mind, we can formulate the following optimization problem:

Test Suite Minimization Problem.

Given: A test suite TS generated for a machine M with $E = \{e_1, e_2, \dots, e_n\}$ the set of events, and subsets of TS , T_i s, one associated with each of the e_i s such that any one of the test cases t_j belonging to T_i can be used to cover e_i .

Problem: Find minimal test suite T from TS which covers all e_i .

As also mentioned in the previous section, this problem is NP-complete because it can be reduced to the minimum set-cover problem as follows. We recall that for us a test case $tc \in TS$ is an execution path which consists in a sequence of events from E . Let be $cov(tc) = \{e \in E | tc \text{ covers } e\}$ the set of events covered by test case tc . By definition, $cov(tc)$ is a subset of E . Therefore the solution T of the above test suite minimization problem is exactly a minimum set cover for E , because

$$\bigcup_{t \in T} cov(t) = E$$

and T is the minimal subset of TS which covers E .

Many solutions have been proposed to solve this test suite minimization problem. Due to its exponential complexity, we use Multi-Objective Evolutionary Algorithms for solving it. For that, we mathematically reformulate it as a constraint bi-objective test suite optimization problem (see TSO1 problem below).

13.5.2 Optimization Criteria

Based on practical experience at SAP, we propose here different test suite optimization criteria.

TSO1-Minimizing the size of the test suite. Due to the restrictions of time, obtaining a minimal test suite which achieves maximal level of coverage is of particular interest among testers. Therefore the goal of this problem is to produce a test suite that contains the smallest possible number of test cases that achieve the same coverage (in our case, the event coverage) as the complete test suite. We formulate this problem as a constraint bi-objective optimization problem: maximize event coverage (the first objective) by a minimum number of test cases (the second objective) under the constraint that at least a test case has been selected. The problem can be mathematically described in the following manner.

Let be $TS = \{t_1, t_2, \dots, t_m\}$ the initial set of m test cases and $E = \{e_1, e_2, \dots, e_n\}$ the set of the events to be covered. We recall that $cov(tc)$ is the set of events covered by the test case tc . Given an order between the elements of a set, a subset $T \subset TS$ can be mathematically represented by a binary vector $x = (x_1, x_2, \dots, x_m) \in \{0, 1\}^m$ with

$$x_i = \begin{cases} 1, & t_i \in T \\ 0, & t_i \notin T \end{cases}, 1 \leq i \leq m.$$

Therefore the constraint bi-objective test suite optimization problem to be solved is the following:

$$\text{Minimize } (f_1(x), f_2(x))$$

Subject to:

$$\sum_{i=1}^m x_i \geq 1 \quad (T \neq \emptyset)$$

Where:

$$f_1(x) = 1 - \sum_{i=1}^m (x_i \cdot \frac{|cov(t_i)|}{n}) \quad (\text{maximize the coverage})$$

$$f_2(x) = \frac{\sum_{i=1}^m x_i}{m} \quad (\text{minimize the size of test suite}).$$

A Pareto-optimal solution of the above problem corresponds to a minimal subset of the test suite TS which achieves a maximal level of coverage. More, we can see that $f_1 : \{0, 1\}^m \rightarrow [0, 1]$ and $f_2 : \{0, 1\}^m \rightarrow (0, 1]$. Therefore we avoid to select the empty set as a solution.

TSO2-Minimizing the number of the executed events. In order to reduce the effort of the testing process, the number of executed events from the whole test suite should be minimized. Therefore we want to obtain test suites which achieve the event coverage criterion with a minimum number of executed events. The first objective function f_1 and the constraint from the problem TSO1 remain valid. Let be $len(tc)$ the length of the test case $tc \in TS$. The second objective function f_2 which can be used to minimize the number of executed events by the subset $T \subset TS$ is

$$f_2(x) = \frac{1}{\sum_{k=1}^m len(t_k)} \sum_{i=1}^m (x_i \cdot len(t_i)).$$

TSO3-Minimizing the length of the longest execution path. The longer execution paths are harder to maintain. In this problem we control the lengths of the execution paths by minimizing the length of the longest test case. The mathematical formulation is the following:

$$\text{Minimize } (f_1(x), f_2(x))$$

Where $f_1(x)$ is the same as for TSO1 problem and

$$f_2(x) = \max\{len(t_i) | x_i = 1 \text{ and } 1 \leq i \leq m\}.$$

The second objective function f_2 is used for minimizing the length of the longest test case.

TSO4-Minimizing the execution time. We measure the execution time for each test case tc from the initial test suite TS . Let us denote by $time(tc)$ the execution time of tc . Then the execution time of a test suite $T \subset TS$ is $\sum_{tc \in T} time(tc)$. In this problem the goal is to minimize the execution time of the test suites. The first objective and the constraint are the same as for TSO1 problem. The second objective function f_2 to be minimized is

$$f_2(x) = \sum_{i=1}^m (x_i \cdot time(t_i)) \text{ (minimize the execution time).}$$

TSO5-Maximizing the distribution quality. In order to understand the problem proposed here, let us consider a simple example. Let be $T_1 = \{e_1e_3e_4, e_1e_2, e_3e_2e_5\}$ and $T_2 = \{e_2e_2e_4, e_1e_2, e_3e_5\}$ two test suites which cover the set of events $E = \{e_1, e_2, \dots, e_5\}$. The events e_1 and e_2 are executed an equal number of times in T_1 , while they are not in T_2 . We say that T_1 has a better *distribution quality*. Therefore the goal is to obtain test suites with a good distribution of the events. This property is a practical requirement of users.

In the following, we propose an objective function which measures the distribution quality of a given test suite $T \subset TS$. Let be $TS = \{t_1, t_2, \dots, t_m\}$ the initial test suite and $E = \{e_1, e_2, \dots, e_n\}$ the set of the events. Let be a matrix A which captures the events covered by each test case tc in TS ; the number of rows of A equals the number of events to be covered, n , and the number of columns equals the number of test cases in the initial test suite, m . Therefore the entries $(a_{ij})_{1 \leq i \leq n, 1 \leq j \leq m}$ of A are

$$a_{ij} = \begin{cases} k, & t_j \text{ covers } e_i \text{ by } k \text{ times} \\ 0, & e_i \text{ is not covered by } t_j \end{cases}, 1 \leq i \leq n, 1 \leq j \leq m.$$

Let be $x = (x_1, x_2, \dots, x_m) \in \{0, 1\}^m$ the mathematical representation of the test suite $T \subset TS$. We define the matrix $D(x)$ to be

$$D(x) = A \times \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix}$$

More exactly, $D(x)$ is a vector of n components $d_i(x), 1 \leq i \leq n$. From the definition, the entry $d_i(x) = \sum_{k=1}^m (a_{ik} \cdot x_k)$ of D denotes the number of times the event e_i was covered by the test suite T .

Now the mean amount of executions per event in T is exactly

$$m_T(x) = \frac{1}{n} \sum_{i=1}^n d_i(x).$$

If the test suite T has a good distribution of the events, we would expect $d_i(x), 1 \leq i \leq n$ values to stay near the mean value $m_T(x)$. Therefore in order to obtain a good distribution of the events we define the objective function to be minimized in the following manner:

$$f(x) = \frac{1}{n} \sum_{i=1}^n (d_i(x) - m_T(x))^2.$$

Let us illustrate this definition on our simple example. We consider that $TS = T1 \cup T2 = \{e_1e_3e_4, e_1e_2, e_3e_2e_5, e_2e_2e_4, e_1e_2, e_3e_5\}$. Then, $x_1 = (1, 1, 1, 0, 0, 0)$ and $x_2 = (0, 0, 0, 1, 1, 1)$ are the mathematical descriptions of T_1 and T_2 respectively. Given that, the matrix A will be

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 2 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

and

$$D(x_1) = A \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 1 \\ 1 \end{pmatrix}, \quad D(x_2) = A \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Further calculation shows that $f(x_1) = 0.24$ and $f(x_2) = 0.64$. Therefore the test suite $T1$ has a better distribution of the events.

We formulate this problem as a constraint single-objective optimization problem and search for solutions which minimize $f(x)$ subject to

$$d_i(x) \geq 1, \quad 1 \leq i \leq n \text{ (each event is covered at least one time).}$$

TSO6-Balancing the lengths while minimizing the longest path.

Finally, we propose here to balance the lengths of the execution paths while we keep valid the two objectives of **TSO3** problem (achieve event coverage while minimize the length of the longest path). Therefore this problem is a 3-objective test suite optimization problem. We search here for test suites which achieve event coverage by short and balanced execution paths. The third objective function can be mathematically formulated as below.

We remember that $len(tc)$ denotes the length of the test case tc . Let be $T \subset TS$ a test suite and x its mathematical description. First, we define the mean of the lengths as

$$m_T^{len}(x) = \frac{1}{|T|} \sum_{i=1}^m (x_i \cdot len(x_i)).$$

If the test suite T contains balanced execution paths, the $len(tc), tc \in T$ values will stay near the mean value $m_T^{len}(x)$. Given that, the third objective function to be minimized can be defined as

$$f_3(x) = \frac{1}{|T|} \sum_{i=1}^m (x_i \cdot (len(t_i) - m_T^{len}(x))^2)$$

We solve all these six test suite optimization problems using multi-objective evolutionary algorithms. In Table 13.6 we summarize the properties of our problems.

13.5.3 Experiments

We provide now the results of a couple of experiments to verify the efficiency and effectiveness of the presented methods.

Solution Encodings. We chose two modern and widely used Pareto efficient genetic algorithms, NSGA-II and SPEA-2 [ZLT01]. When using evolutionary algorithms for solving a multi-objective test suite optimization problem, we must properly encode the possible solutions of the problem. Let be $T \subset TS$ a subset of the initial test suite $TS = \{t_1, t_2, \dots, t_m\}$. We use the mathematical representation $x \in \{0, 1\}^m$ of T (see Section 13.5.2) to encode the possible solutions. Therefore binary encoding is considered to be a natural representation for the possible solutions. The inclusion and

Table 13.6: Summarize the six test suite optimization problems.

Problem	Type	Constraint	Description
TSO1	bi-objective	yes	Minimizing the size of the test suite
TSO2	bi-objective	yes	Minimizing the no. of the executed events
TSO3	bi-objective	no	Minimizing the longest execution path
TSO4	bi-objective	yes	Minimizing the execution time
TSO5	single-obj.	yes	Maximizing the distribution quality
TSO6	3-objective	no	Balancing the lengths + TSO3 problem

exclusion of a test case within a subset of the initial test suite are represented by 1 and 0 respectively in a binary string (*chromosome* string).

Subjects. We conducted the experiments with a total of five test suite subjects of varying sizes and complexity levels. The test suites were generated from two industrial inspired Event-B models: the BepiColombo and SSFPilot models which are publicly available DEPLOY model repository¹². The first four machines are different levels of refinements of BepiColombo project and the last machine is the high level of abstraction of SSFPilot model. The sizes of the machines are listed in Table 13.7. Moreover, the test suite generated from these Event-B models were obtained using the MBT plugin¹³ available for Rodin. The test generation algorithm is based on [Ipa11] and described in Section 13.4.

The two Event-B models are summarized below:

- *BepiColombo*: This is an abstract model¹⁴ of two communication modules in the embedded software on a space craft. The Event-B model was proposed for formal validation of software parts of BepiColombo mission to Mars¹⁵. The model has different levels of refinements. In the abstraction, M_0 , the main goal of the system is modeled. The details of the system are added through three refinement levels, M_1 , M_2 and M_3 . The modeling approach starts on the first level with 5 set-type variables and 5 events and ends up with 18 variables and 16 events.

¹²<http://deploy-eprints.ecs.soton.ac.uk>

¹³http://wiki.event-b.org/index.php/MBT_plugin

¹⁴http://eprints.ecs.soton.ac.uk/22048/5/Rodin_Space_Craft.zip

¹⁵See http://deploy-eprints.ecs.soton.ac.uk/72/1/BepiColombo_-_Modelling_Approach.pdf and <http://en.wikipedia.org/wiki/BepiColombo>

Table 13.7: Sizes of five test suite subjects generated from two industrial inspired models (number of events, size of test suites and maximum length of test cases).

Subject	No. of events	Size of TS	Max. size of tcs
BepiColombo_M0	5	40	7
BepiColombo_M1	10	170	7
BepiColombo_M2	12	256	7
BepiColombo_M3	16	240	7
SSFPilot_TCTM	13	786	8

- *SSFPilot*: This is an Event-B model¹⁶ of a pilot for a complex on-board satellite mode-rich system: Attitude and Orbit Control System (AOCS). In [ITL⁺10a] the authors present a formal development of an AOCS in Event-B modeling language. They show that refinement in Event B provides the engineers with a scalable formal technique that enables both development of mode-rich systems and proof-based verification of their mode consistency.

Results. The test suite optimization techniques attempt to reduce the test suite cost w.r.t. a given coverage criterion (event coverage in our case). Given that, the percentage reduction will be used as a measure for comparative analysis. To increase the confidence, we compare the results produced by the two algorithms: NSGA-II and SPEA-2.

We have used the multi-objective evolutionary algorithm framework jMetal [DN11] for our experiments. The two algorithms were configured with population size of 100. The archive size of SPEA-2 was set to the same value, 100. The stopping criterion is to reach the maximum number of generation which was set to 100. The both algorithms use the following genetic operators: *the binary tournament selection* operator, *the single point crossover* operator with probability of 0.9 and *the single bit-flip mutation* operator with the mutation rate of $1/m$ where m is the length of the bit-string (i.e. the size of the initial test suite).

For each test suite subject, each optimization problem and each algorithm, 100 independent runs were performed. The results are presented in Tables 13.8-13.13. To compare the results, we computed for each problem the specific objective function values for the initial test suite. For example,

¹⁶<http://deploy-eprints.ecs.soton.ac.uk/58/>

Table 13.8: TSO1. Average reduced sizes for optimized test suite T .

		NSGA-II		SPEA-2	
Subject	$f_2(x_{TS})$	Avg $f_2(x_T)$	Avg%	Avg $f_2(x_T)$	Avg%
BepiColombo_M0	40	1.03	97.42	1.01	97.47
BepiColombo_M1	170	7.59	95.53	8.72	94.87
BepiColombo_M2	256	28.87	88.72	30.98	87.89
BepiColombo_M3	240	26.14	89.10	27.97	88.34
SSFPilot_TCTM	786	228.42	70.93	232.5	70.41

Table 13.9: TSO2. Average reduced number of executed events for optimized test suite T .

		NSGA-II		SPEA-2	
Subject	$f_2(x_{TS})$	Avg $f_2(x_T)$	Avg%	Avg $f_2(x_T)$	Avg%
BepiColombo_M0	252	8.02	96.8	8.02	96.8
BepiColombo_M1	1300	65.09	94.99	71.93	94.46
BepiColombo_M2	1977	224.42	88.65	236.34	88.04
BepiColombo_M3	1873	204.77	89.06	221.39	88.17
SSFPilot_TCTM	6554	1897.79	71.04	1931.98	70.52

Table 13.10: TSO3. Average length of the longest path of optimized test suite T .

Subject	NSGA-II	SPEA-2
BepiColombo_M0	4.69	4.84
BepiColombo_M1	7	7
BepiColombo_M2	7	7
BepiColombo_M3	7	7
SSFPilot_TCTM	8	8

Table 13.11: TSO4. Average execution time (in seconds) of optimized test suite T .

		NSGA-II		SPEA-2	
Subject	$f_2(x_{TS})$	Avg $f_2(x_T)$	Avg%	Avg $f_2(x_T)$	Avg%
BepiColombo_M0	4.6	0.13	97.07	0.14	96.95
BepiColombo_M1	48.43	1.88	96.11	2.16	95.54
BepiColombo_M2	130.16	12.39	90.48	13.40	89.70
BepiColombo_M3	204.28	20.43	89.99	22.13	89.16
SSFPilot_TCTM	197.80	50.78	74.32	51.38	74.02

Table 13.12: TSO5. Average distribution quality of optimized test suite T .

		NSGA-II		SPEA-2	
Subject	$f(x_{TS})$	Avg $f(x_T)$	Avg%	Avg $f(x_T)$	Avg%
BepiColombo_M0	520.24	0.16	99.96	0.16	99.96
BepiColombo_M1	8771.4	17.03	99.80	22.45	99.74
BepiColombo_M2	19840.90	238.98	98.79	270.26	98.63
BepiColombo_M3	14432.43	169.14	98.82	191.42	98.67
SSFPilot_TCTM	166187.40	13251.76	92.02	13667.67	91.77

Table 13.13: TSO6. Average balancing values of the lengths of optimized test suite T .

		NSGA-II		SPEA-2	
Subject	$f_3(x_{TS})$	Avg $f_3(x_T)$	Avg%	Avg $f_3(x_T)$	Avg%
BepiColombo_M0	2.16	0.00	100	0.00	100
BepiColombo_M1	1.81	0.21	88.27	0.22	87.52
BepiColombo_M2	1.57	0.33	78.41	0.34	77.87
BepiColombo_M3	1.62	0.36	77.76	0.37	77.04
SSFPilot_TCTM	2.21	1.15	47.96	1.17	47.05

the column $f_3(x_{TS})$ from the Table 13.13 indicates the values of the third objective function of the problem TSO6 when computed for the initial test suite TS . Otherwise, in each table, the average values of specific objective functions of the solutions are indicated. As shown in the tables, the results of the two algorithms are comparable. We obtained high values for the percentage reduction of test suite because of the simplicity of the event coverage criterion.

13.5.4 Conclusions

In this subsection the multi-objective test suite optimization problem for Event-B testing was introduced. Different optimization criteria were proposed and the resulted problems were solved using two modern multi-objective evolutionary algorithms. For all optimization problems the considered test adequacy criterion was the event coverage. All our optimization problems can be easily formulated in a more general framework: a test suite T must meet a set of n requirements $\{r_1, r_2, \dots, r_n\}$ to provide the desired 'adequate' testing of the model. We will consider in the future more complex coverage criteria.

13.6 Conclusion

In this chapter, we described several methods of test generation from Event-B models. The methods are supported by implementations in form of plugins available for the Rodin platform and extensive experiments were performed.

Bibliography

- [ABC⁺02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings FATES'02*, pages 105–120, August 2002. Technical Report, INRIA.
- [ABH⁺10] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An open toolset for modelling and reasoning in event-B. *STTT*, 12(6):447–466, 2010.
- [ABHV06] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *ICFEM 2006*, 2006.
- [Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [Abr06] J.-R. Abrial. Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th international conference on Software engineering*, pages 761–768, 2006.
- [Abr09] J.-R. Abrial. Event model decomposition. Technical Report 626, ETH Zurich, May 2009.
- [Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, UK, 2010.
- [AC04] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: Feature modeling plug-in for eclipse. In *Eclipse '04: Proceedings of the 2004 OOPSLA*, pages 67–72, NY, USA, 2004. ACM.
- [AC05] J.-R. Abrial and D. Cansell. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS*, 2005.

- [AH07] J.-R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.
- [AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. ICSE '09, pages 221–231, USA, 2009.
- [AL93] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, Vol.15, No.1, pp.73–132, 1993.
- [AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 1995.
- [ALR01] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental Concepts of Dependability. 2001. Research Report No 1145, LAAS-CNRS.
- [ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
- [Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
- [ASAA08] Idir Aït-Sadoune and Yamine Aït-Ameur. Animating Event B models by formal data models. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 37–55. Springer-Verlag, 2008.
- [Bac90] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.
- [Bar85] H. Barringer. *A Survey of Verification Techniques for Parallel Programs*. LNCS. Springer, 1985.
- [Bar06] J. Barnes. *Programming in Ada 2005*. International Computer Science Series. Addison Wesley, 2006.

- [Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *SPLC '05: Proceedings of the 9th International Software Product Line Conference*, pages 7–20. Springer, 2005.
- [BB06] Elisabeth Ball and Michael Butler. Using Decomposition to Model Multi-agent Interaction Protocols in Event-B. In *FM'06 Doctoral Symposium*. Springer, 2006.
- [BB09] Elisabeth Ball and Michael Butler. Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction. In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 104–129. 2009.
- [BBP⁺03] D. Bert, S. Boulmé, M. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In *FME*, pages 94–113, 2003.
- [BDSW11] P. Boström, F. Degerlund, K. Sere, and M Waldén. Concurrent Scheduling of Event-B Models. In *Proceedings 15th International Refinement Workshop*, pages 166–182, June 2011.
- [BDV04] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Lett.*, XXIV:1–74, June 2004.
- [Ber07] G. Berry. Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In *FMICS*, page 2, 2007.
- [BFJ⁺10] M. Butler, J. Fitzgerald, M. Jastram, C. Jones, L. Laibinis, M. Mazzara, A. Rezazadeh, and K. Sere. Methods: Tackling industry specific challenges (d31). Technical Report D31, DEPLOY Project, 2010.
- [BFRR09] Jeremy W. Bryans, John S. Fitzgerald, Alexander Romanovsky, and Andreas Roth. Formal modelling and analysis of business information systems with fault tolerant middleware. In *Proceedings 14th IEEE International Conference on Engineering of Complex Computer Systems ICECCS 2009*, pages 68–77. IEEE, 2009.

- [BFRR10] J. W. Bryans, J. S. Fitzgerald, A. Romanovsky, and A. Roth. Patterns for Modelling Time and Consistency in Business Information Systems. In *Proc. 15th IEEE Intl. Conf. on Engineering of Complex Computer Systems*, pages 105–114, March 2010.
- [BGJ06] Denis Besnard, Cristina Gacek, and Clifford B. Jones. *Structure for dependability - computer-based systems from an interdisciplinary perspective*. Springer, 2006.
- [BGL⁺97] M. Butler, J. Grundy, T. Långbacka, R. Rukšenas, and J. von Wright. The Refinement Calculator: Proof Support for Program Refinement. *Proc. of Formal Methods Pacific*, 1997.
- [Bit02] K. Bittner. *Use Case Modeling*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT press, 2008.
- [BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralisation of Process Nets with Centralised Control. In *Proc. of the 2nd ACM SIGACTS-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.
- [BL11] Jens Bendisposto and Michael Leuschel. Automatic flow analysis for Event-B. In *Proc. of FASE’11*, volume 6603 of *LNCS*, pages 50–64. Springer, 2011.
- [BLLS08] Jens Bendisposto, Michael Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour RODIN. *Technique et Science Informatiques*, 27(8):1065–1084, 2008.
- [BM99] L. Burdy and J.-M. Meynadier. Automatic Refinement. *Workshop on Applying B in an industrial context : Tools, Lessons and Techniques - Toulouse, FM’99*, 1999.
- [Bos00] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [Bos10] P. Boström. Creating Sequential Programs from Event-B Models. In *Proceedings of the 8th international conference on Integrated Formal Methods*, IFM’10, pages 74–88, Berlin, Heidelberg, 2010. Springer-Verlag.

- [BP05] D. Bert and M. Potet. GeneSyst: A Tool to Reason About Behavioral Aspects of B Event Specifications, Application to Security Properties. In *LNCS, vol. 3455*, pages 299–318. Springer, 2005.
- [BS89] R-J. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comp. Prog.*, 1989.
- [BS91] R-J. Back and K. Sere. Superposition refinement of parallel algorithms. In *FORTE*, 1991.
- [BS03] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [Bur99] A. Burns. The Ravenscar Profile. *Ada Lett.*, XIX:49–52, December 1999.
- [But96a] Michael J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, 1996.
- [But96b] R.W. Butler. *An introduction to requirements capture using PVS: Specification of a simple autopilot*. Technical report, NASA TM-110255, May 1996.
- [But97] Michael Butler. An Approach to the Design of Distributed Systems with B AMN. In *Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, pages 221–241, 1997.
- [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.
- [But04] B. Buth. Analysing mode confusion: An approach using fdr2. In *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP’04)*, pages 101–114, 2004.
- [But06] M. Butler. Synchronisation-based Decomposition for Event-B. In: RODIN Deliverable D19 Intermediate report on methodology. Technical report, University of Southampton, 2006.

- [But09a] M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009*, Springer, LNCS 5423, volume LNCS. Springer, February 2009.
- [But09b] M. Butler. Incremental Design of Distributed Systems with Event-B. In *Engineering Methods and Tools for Software Safety and Security*. IOS Press, 2009.
- [But09c] Michael Butler. Chapter 8: Modelling Guidelines for Discrete Control Systems, Deploy Deliverable D15, D6.1 Advances in Methods Public Document. <http://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf>, 2009. cited 2011 Jan.
- [But09d] Michael Butler. Decomposition Structures for Event-B. In Michael Leuschel and Heike Wehrheim, editors, *IFM*, volume 5423 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2009.
- [But09e] Michael Butler. Towards a cookbook for modelling and refinement of control problems. 2009.
- [Bv90] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1990.
- [BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Texts in Computer Science. Springer-Verlag New York, Inc., 1998.
- [CAK⁺05] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. Model-driven software product lines. In *OOPSLA '05*, pages 126–127, NY, USA, 2005. ACM.
- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

- [CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: a requirements engineering perspective. In *Proceedings of FASE'08/ETAPS'08*, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.
- [CJ07] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.
- [CKE03] Muffy Calder, Mario Kolberg, and B Evan H. Magill. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41:115–141, 2003.
- [Clea] ClearSy. B Language Reference Manual version 1.8.6. <http://www.atelierb.eu/ressources/manrefb1.8.6.uk.pdf>.
- [Cleb] ClearSy System Engineering. *The B Language Reference Manual*, version 4.6 edition.
- [CMR07] Dominique Cansell, Dominique Méry, and Joris Rehm. Time constraint patterns for Event-B development. In Julliand and Kouchnarenko [JK06], pages 140–154.
- [CN01] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [CPS06] Cezar Câmpeanu, Andrei Paun, and Jason R. Smith. Incremental construction of minimal deterministic finite cover automata. *Theoret. Comput. Sci.*, 363(2):135–148, 2006.
- [CSK] CSK Systems Corporation. The VDM++ Language Manual.
- [DAPM00] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Proceedings of the Parallel Problem Solving from Nature VI Conference*, volume 1917 of *LNCS*, pages 849–858. Springer, 2000.
- [DDMvL97] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 612–613, New York, NY, USA, 1997. ACM.

- [Del97] C. Dellarocas. The SYNTHESIS Environment for Component-Based Software Development. In *8th International Workshop on Software Technology and Engineering Practice (STEP '97)*, 1997.
- [Depa] DEPLOY Deliverable D20 – Report on Pilot Deployment in the Space Sector. FP7 ICT DEPLOY Project. <http://www.deploy-project.eu/> (Accessed January 2010).
- [Depb] Deploy Code Generation Wiki Site. at http://wiki.event-b.org/index.php/Code_Generation_Activity.
- [DGS11] F. Degerlund, R. Grönblom, and K. Sere. Code Generation and Scheduling of Event-B Models. Technical Report 1027, Turku Centre for Computer Science, 2011.
- [Die96] Henning Dierks. The production cell: A verified real-time system. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 208–227. Springer, 1996.
- [Dij75] E.W. Dijkstra. Guarded Commands, Non-determinacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.
- [Din11] Ionut Dinca. Multi-objective test suite optimization for Event-B models. In *Proc. of Int. Conf. on Informatics Engineering and Information Science (ICIEIS'11)*, volume 251 of *CCIS*, pages 551–565. Springer, 2011.
- [DIRR09] Fernando Luís Dotti, Alexei Iliasov, Leila Ribeiro, and Alexander Romanovsky. Modal systems: Specification, refinement and realisation. In *ICFEM*, pages 601–619, 2009.
- [DN11] Juan José Durillo and Antonio J. Nebro. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011.
- [DP01] Nachum Dershowitz and David A. Plaisted. Rewriting. In *Handbook of Automated Reasoning*, pages 535–610. Elsevier and MIT Press, 2001.

- [dRdBH⁺01] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. CUP, 2001.
- [DSI⁺11] Ionut Dinca, Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Test data generation for Event-B models using genetic algorithms. In *Proc. of 2nd International Conference on Software Engineering and Computer Systems (ICSECS'11)*, volume 181 of *CCIS*, pages 76–90. Springer, 2011.
- [DW07] Gero Decker and Mathias Weske. Local enforceability in interaction Petri Nets. In *Proceedings of the 5th International Conference on Business Process Management (BPM'07)*, volume 4714 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2007.
- [DWS07] F. Degerlund, M. Waldén, and K. Sere. Implementation Issues Concerning the Action Systems Formalism. In *Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT '07*, pages 471–479, Washington, DC, USA, 2007. IEEE Computer Society.
- [EB06] Neil Evans and Michael Butler. A Proposal for Records in Event-B. In Tobias Nipkow, Jayadev Misra, and Emil Sekerinski, editors, *Formal Methods 2006*, volume LNCS 4, pages 221–235. Springer, lncs, 2006.
- [EB08] A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.
- [EB10] A. Edmunds and M. Butler. Tool Support for Event-B Code Generation, 2010.
- [EB11] A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.

- [Edi] Cliff Jones Editor. Deploy deliverable d15: Advances in methodological wps. Technical report, <http://www.deploy-project.eu/pdf/D15final.pdf>.
- [Edm10] A. Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, March 2010.
- [EOL] Eclipse Object Language. <http://www.eclipse.org/gmt/epsilon/doc/eol/>.
- [ERa] A. Edmunds and A. Rezazedah. Event-B Project Archives: Tasking Event-B Tutorial. University of Southampton. at <http://deploy-eprints.ecs.soton.ac.uk/304/>.
- [ERb] A. Edmunds and A. Rezazedah. Event-B Wiki: Development of a Heating Controller System. at http://wiki.event-b.org/index.php/Development_of_a_Heating_Controller_System.
- [EU-] EU-project DEPLOY. online at <http://www.deploy-project.eu/>.
- [Eve] Event-B.org. Generic instantiation. http://wiki.event-b.org/index.php/Generic_Instantiation.
- [Fed05] Federal Motor Carrier Safety Administration. Concept of operations and voluntary operational requirements for lane departure warning systems (LDWS) on-board commercial motor vehicles. <http://www.fmcsa.dot.gov/facts-research/research-technology/report/lane-departure-warning-systems.htm>, 2005. cited 2011 Jan.
- [Fel67] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, 1967.
- [FGK⁺10] Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors. *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*. Springer, 2010.
- [FHP⁺05] A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann, and D. Wild. Concretization and formalization of

- requirements for automotive embedded software systems development. In *The Tenth Australian Workshop on Requirements Engineering (AWRE)*, pages 60–65, 2005.
- [Fit91] J. Fitzgerald. Modularity in Model-oriented Formal Specifications and its Interaction with Formal Reasoning. University of Manchester, 1991. Ph.D. Thesis.
- [FMEa] FMEA Info Centre. <http://www.fmeainfocentre.com/>.
- [FMEb] FMEA Info Centre. <http://www.fmeainfocentre.com/>.
- [FP09] Brett Stanley Feero and Partha Pratim Pande. Networks-on-Chip in a Three-Dimensional Environment: A Performance Evaluation. *IEEE Transactions on Computers*, 58:32–45, 2009.
- [FRB11] Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael Butler. Applying atomicity and model decomposition to a space craft system in Event-B. In *Proc. of 3rd NASA Formal Methods (NFM’11)*, volume 6617 of *LNCS*, pages 328–342. Springer, 2011.
- [Fut08] M. J. Futch. *Leibniz’s Metaphysics of Time and Space*. Springer-Verlag GmbH, 2008.
- [GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering (ICRE’94)*, pages 94–101. IEEE Computer Society Press, 1994.
- [GGJZ00] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *Software, IEEE*, 17(3):37–43, 2000.
- [GGL⁺11] K. Grau, R. Gmehlich, F. Loesch, J.C. Deprez, R. De Landtsheer, and C. Ponsard. Report on enhanced deployment in the automotive sector (d31). Technical Report D38, DEPLOY Project, 2011.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley. ISBN 0-201-63361-2, 1995.
- [GJGZ00] C. A. Gunter, M. Jackson, E. L. Gunter, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17:37–43, 2000.

- [GJSB04] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification - Third Edition*. Addison-Wesley, 2004.
- [GKP09] T. Gangadharappa, M. Koop, and D. K. Panda. Designing and Evaluating MPI-2 Dynamic Process Management Support for InfiniBand. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICCPPW '09, pages 89–96, Washington, DC, USA, 2009. IEEE Computer Society.
- [GM93] Mike J. C. Gordon and Thomas F. Melham. *Introduction to HOL*. Cambridge, 1993.
- [GPB11] Ali Gondal, Michael Poppleton, and Michael Butler. Capability & potential for formal feature-oriented reuse in event-b. (*To appear in*) *Information and Software Technology*, 0, 2011.
- [GPBS10] Ali Gondal, Michael Poppleton, Michael Butler, and Colin Snook. Feature-Oriented Modelling Using Event-B. In *SETP-10*, Orlando, FL, USA, 2010.
- [GPIS04] Cristian Grecu, Partha Pratim Pande, André Ivanov, and Res Saleh. A Scalable Communication-Centric SoC Interconnect Architecture. In *Proceedings of the 5th International Symposium on Quality Electronic Design*, ISQED '04, pages 343–348, Washington, DC, USA, 2004. IEEE Computer Society.
- [GPS09] A. Gondal, M. Poppleton, and C. Snook. Feature composition – towards product lines of Event-B models. In *MDPLE 2009: 1st International Workshop on Model-Driven Product Line Engineering*, Twente, The Netherlands, June 2009. In press.
- [GR01] Angelo Gargantini and Elvinia Riccobene. ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.
- [Grö09] R. Grönblom. A Framework for Code Generation and Parallel Execution of Event-B Models. Master’s thesis, Åbo Akademi University, 2009.
- [Gru96] J. Grundy. Transformational Hierarchical Reasoning. *The Computer Journal*, 39(4):291–302, 1996.
- [GSLB03] G.Karsai, J. Sztipanovits, Á. Lédeczi, and T. Bapty. Model-integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

- [GSLG04] Rajesh Gupta, Sandeep Kumar Skuhla, and Paul Le Guernic. *Formal Methods and Models for System Design: A System Level Perspective (The Kluwer international series in video computing)*. Springer, 1 edition, October 2004.
- [HA10] Thai Son Hoang and Jean-Raymond Abrial. Event-b decomposition for parallel programs. In Frappier et al. [FGK⁺10], pages 319–333.
- [Hal07] Stefan Hallerstede. Justifications for the Event-B Modelling Notation. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *LNCS*, pages 49–63. Springer, 2007.
- [Hal08] Stefan Hallerstede. On the Purpose of Event-B Proof Obligations. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 125–138, Berlin, Heidelberg, 2008. Springer-Verlag.
- [Hal09a] Stefan Hallerstede. Proving quicksort correct in Event-B. *Electr. Notes Theor. Comput. Sci.*, 259:47–65, 2009.
- [Hal09b] Stefan Hallerstede. Proving Quicksort correct in Event-B. In *Refine 2009*, ENTCS, 2009.
- [Hal10] S. Hallerstede. Structured Event-B models and proofs. In *Abstract State Machines, B and Z*, volume 5977 of *LNCS*, pages 273–286. Springer-Verlag, 2010.
- [Han72] P.B. Hansen. Structured multiprogramming. *Commun. ACM*, 15:574–578, July 1972.
- [Han95] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Elsevier, 1995.
- [Har03] John Harrison. Formal Verification at Intel. *Logic in Computer Science, Symposium on*, 0:45, 2003.
- [HBA09] Thai Son Hoang, David Basin, and Jean-Raymond Abrial. Development of access control systems in Event-B. Technical Report 624, Department of Computer Science, ETH Zurich, june 2009. Available at <http://www.inf.ethz.ch/research/disstechreps/techreports>.

- [Hef09] Randy Heffner. Across all vertical industry groups, the majority of SOA users are expanding its use. Research report, Forrester Research, May 2009.
- [Heh93] E. C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
- [Hen96] T.A. Henzinger. The Theory of Hybrid Automata. In *LICS*, pages 278–292, 1996.
- [HFA09a] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-b patterns and their tool support. In Dang Van Hung and Padmanabhan Krishnan, editors, *SEFM*, pages 210–219. IEEE Computer Society, 2009.
- [HFA09b] Thai Son Hoang, Andreas Furst, and Jean-Raymond Abrial. Event-b patterns and their tool support. In *SEFM*, pages 210–219, 2009.
- [HH07] S. Hallerstede and T. S. Hoang. Qualitative probabilistic modelling in Event-B. In J. Davies and J. Gibbons, editors, *IFM 2007, Integrated Formal Methods*, pages 293–312, 2007.
- [HHS86] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Wilhelm, editors, *Proceedings ESOP’86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.
- [HJ98] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
- [HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.
- [HJK⁺00] A. Hemani, A. Jantch, K. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on a Chip: An architecture for billion transistor era. In *IEEE NorChip Conference*, 2000.
- [HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5:231–261, July 1996.

- [HL96] M. Heimdahl and N. Leveson. Completeness and Consistency in Hierarchical State- Based Requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996.
- [HL09] S. Hallerstede and M. Leuschel. How to explain mistakes. In Jeremy Gibbons and José Nuno Oliveira, editors, *Teaching Formal Methods, Second International Conference, TFM 2009, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5846 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2009.
- [HLP11] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Validation of formal models by refinement animation. *Science of Computer Programming*, In Press, 2011.
- [HM03] J. Hu and R. Marculescu. Energy-aware mapping for tile-based NOC architectures under performance constraints. In *Asia South Pacific Design Automation Conference*, pages 233–239, 2003.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 1969.
- [Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, 1974.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [Hoa09] T.S. Hoang. Findp development using decomposition. <http://deploy-eprints.ecs.soton.ac.uk/154/>, 2009.
- [HW03] K. Hinderer and T.-H. Waldmann. The Critical Discount Factor for Finite Markovian Decision Processes with an Absorbing Set. In *Mathematical Methods fo Operations Research*, pages 1–19. Springer-Verlag, 2003.
- [IDS] Florentin Ipate, Ionut Dinca, and Alin Stefanescu. Model learning and test generation for Event-B using cover automata. To be submitted. The current draft can be found online at: <http://dl.dropbox.com/u/12344140/model-learning-test.pdf>.
- [Ili08a] A. Iliasov. Finer Plugin. <http://finer.iliasov.org>, 2008.

- [Ili08b] Alexei Iliasov. *Design Components*. PhD thesis, School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU, United Kingdom, 2008.
- [Ili09] A. Iliasov. On Event-B and control flow. Technical Report CS-TR-1159, School of Computing Science, Newcastle University, 2009.
- [ILT⁺11] Alexei Iliasov, Linas Laibinis, Elena Troubitsyna, Alexander Romanovsky, and Timo Latvala. Augmenting Event B Modelling with Real-Time Verification. In *Technical Report 1006, Turku Centre for Computer Science*, 2011.
- [Ipa10] Florentin Ipate. Bounded sequence testing from deterministic finite state machines. *Theoret. Comput. Sci.*, 411(16-18):1770–1784, 2010.
- [Ipa11] Florentin Ipate. Learning finite cover automata from queries. *Journal of Computer and System Sciences*, 2011. In Press. Online at: <http://dx.doi.org/10.1016/j.jcss.2011.04.002>.
- [IRA⁺07] A. Iliasov, A. Romanovsky, B. Arief, L. Laibinis, and E. Troubitsyna. On Rigorous Design and Implementation of Fault Tolerant Ambient Systems. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 141–145, Washington, DC, USA, 2007. IEEE Computer Society.
- [IRD09] Alexei Iliasov, Alexander Romanovsky, and Fernando Luis Dotti. Structuring specifications with modes. *Latin-American Symposium on Dependable Computing*, pages 81–88, 2009.
- [ITL⁺10a] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Developing mode-rich satellite software by refinement in Event B. In *Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS'10)*, volume 6371 of *LNCS*, pages 50–66. Springer, 2010.
- [ITL⁺10b] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting Reuse in Event B Development: Modularisation Approach. In *Proceedings of Abstract State Machines, Alloy, B, and Z (ABZ*

2010), *Lecture Notes in Computer Science*, Vol.5977, pp. 174–188, Springer, 2010.

- [ITL⁺10c] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, and T. Latvala. Verifying Mode Consistency for On-Board Satellite Software. In *Proc. of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP'10)*, pages 126–141, 2010.
- [Jac01] M. Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [Jac10] Michael Jackson. Engineering and software engineering. In Sebastian Nanz, editor, *The Future of Software Engineering*, pages 100–114. Springer, 2010.
- [Jas10] M. Jastram. ProR, an Open Source Platform for Requirements Engineering based on RIF. *SEISCONF*, 2010.
- [JG11a] M. Jastram and A. Graf. Requirements, traceability and DSLs in Eclipse with the requirements interchange format (RIF/ReqIF). In *Tagungsband des Dagstuhl-Workshop MBEES*. fortiss GmbH, München, 2011.
- [JG11b] M. Jastram and Andreas Graf. Requirements Modeling Framework. *Eclipse Magazin*, 6.11, 2011.
- [JHJ07] Cliff B. Jones, Ian J. Hayes, and Michael A. Jackson. Formal methods and hybrid real-time systems. chapter Deriving specifications for systems that are connected to the physical world, pages 364–390. Springer-Verlag, Berlin, Heidelberg, 2007.
- [JHL11] M. Jastram, S. Hallerstede, and L. Ladenberger. Mixing Formal and Informal Model Elements for Tracing Requirements. In *Automated Verification of Critical Systems (AVoCS)*, 2011.
- [JHLJ10] M. Jastram, S. Hallerstede, M. Leuschel, and A. G Russo Jr. An approach of requirements tracing in formal refinement. In *VSTTE*. Springer, 2010.
- [JHLR10] Michael Jastram, Stefan Hallerstede, Michael Leuschel, and Aryldo Russo. An approach of requirements tracing in formal refinement. In *Verified Software: Theories, Tools, Experiments*,

volume 6217 of *Lecture Notes in Computer Science*, pages 97–111. Springer Berlin / Heidelberg, 2010.

- [JK06] Jacques Julliand and Olga Kouchnarenko, editors. *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2006.
- [JM94] Cliff B. Jones and C. A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Inf.*, 31(5):399–430, 1994.
- [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. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.
- [Jon85] C.B. Jones. The role of proof obligations in software design. In *TAPSOFT, V.2*, LNCS, 1985.
- [Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.
- [Jon96] 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.
- [Jon07] C.B. Jones. Splitting atoms safely. *Theor. Comput. Sci.*, 2007.
- [JP10] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 2010.

- [KCH⁺90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990.
- [KGN⁺09] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whitemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing Testing with Formal Verification in Intel Core™ i7 Processor Execution Engine Validation. In *Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09*, pages 414–429, Berlin, Heidelberg, 2009. Springer-Verlag.
- [KJ10] E. Kang and D. Jackson. Dependability arguments with trusted bases. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, page 262–271, 2010.
- [KP09] Stefan Kätker and Sabine Patig. Model-driven development of service-oriented business application systems. In *Business Services: Konzepte, Technologien, Anwendungen*, volume Band 1, pages 171–180. Österreichische Computer Gesellschaft, 2009.
- [KPP08] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The epsilon transformation language. pages 46–60. 2008.
- [KPSD11] Maryam Kamali, Luigia Petre, Kaisa Sere, and Masoud Daneshtalab. Refinement-based modeling of 3d nocs. In *Fundamental of Software Engineering (FSEN), 2011 4th IPM International Conference on*, 20 2011–april 22 2011.
- [KR78] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [KR88] B.W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.
- [KR09] Vitaly Kozyura and Andreas Roth. Generation of gluing invariants for checking local enforceability of message choreographies. In Michael Jastram, Linas Laibinis, Felix Lösch, and Manuel Mazzara, editors, *Proceedings of Deploy Technical Workshop 2009*. Newcastle University, Technical Report, 2009.

- [KRW09] Vitaly Kozyura, Andreas Roth, and Wei Wei. Local enforceability and unconsumable messages in choreography models. In *Proceedings of 4th South-East European Workshop on Formal Methods (SEEFM)*. IEEE Computer Society, 2009.
- [KS60] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand Company, 1960.
- [Kwi07] M. Kwiatkowska. Quantitative Verification: Models Techniques and Tools. In *ESEC/FSE'07*, pages 449–458. ACM, 2007.
- [Lam97] L. Lamport. Composition: A way to make proofs harder. In *COMPOS*, 1997.
- [Lam02] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [LB03] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: International Symposium of Formal Methods Europe*, pages 855–874, 2003.
- [LB05] Michael Leuschel and Michael Butler. Automatic refinement checking for B. In Kung-Kiu Lau and Richard Banach, editors, *Proceedings ICFEM'05*, volume 3785 of *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 2005.
- [LB08] Michael Leuschel and Michael Butler. ProB: An Automated Analysis Toolset for the B Method. *Software Tools for Technology Transfer*, 10(2), 2008.
- [LB10] Igor Loi and Luca Benini. An efficient distributed memory interface for many-core platform with 3D stacked DRAM. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 99–104, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.
- [Leu08] Michael Leuschel. The high road to formal validation. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Proceedings ABZ 2008*, volume 5238 of *Lecture Notes in Computer Science*, pages 4–23, 2008.
- [Lev95] Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

- [LFFP09] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale B models. In *Proceedings FM'2009*, volume 5850 of *LNCS*, pages 708–723. Springer, 2009.
- [LGG⁺10] F. Loesch, R. Gmehlich, K. Grau, C. Jones, and M. Mazzara. Report on Pilot Deployment in Automotive Sector. Technical Report D7, DEPLOY Project, 2010.
- [LGM⁺10] F. Loesch, R. Gmehlich, K. Grau M., Mazzara, and C. Jones. Pilot deployment in the automotive sector (d19). Technical report, DEPLOY Project, 2010.
- [LH03] Wen-Shiu Liao and Pao-Ann Hsiung. Creating a Formal Verification Platform for IBM CoreConnect-based SoC. In *In Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis (ATVA2003)*, pages 7–18, 2003.
- [LHHR94] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.*, 20:684–707, 1994.
- [Lin95] Thomas Lindner. Task description. In Claus Lewerentz and Thomas Lindner, editors, *Formal Development of Reactive Systems*, volume 891 of *LNCS*. Springer, 1995.
- [LIR10] Ilya Lopatkin, Alexei Iliasov, and Alexander Romanovsky. On fault tolerance reuse during refinement. In *2nd International Workshop on Software Engineering for Resilient Systems, SERENE'10*, London, UK, April 2010. available as CS-TR-1188 at Newcastle University, UK.
- [LKL02] Kwanwoo Lee, Kyo Chul Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *ICSR-7*, pages 62–77, UK, 2002. Springer-Verlag.
- [LL94] C Lewerentz and T Lindner. Case study 'production cell': A comparative study in formal specification and verification. In Manfred Broy and Stefan Jähnichen, editors, *Tech. Rep., Forschungszentrum Informatik*, volume 1009 of *LNCS*, pages 1–54. LNCS 891, Springer-Verlag, 1994.
- [Lor] Loria. EB2ALL. <http://eb2all.loria.fr/>.

- [LPS⁺97] N.G. Leveson, L.D. Pinnel, S.D. Sandys, S. Koga, and J.D. Reese. Analyzing Software Specifications for Mode Confusion Potential. In *Proc. of the Workshop on Human Error and system Development*, pages 132–146, 1997.
- [LPT⁺10] Ilya Lopatkin, Yuliya Prokhorova, Elena Troubitsyna, Alexei Iliasov, and Alexander Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems. In *Technical Report 1003*. TUCS, March 2010.
- [LS91] K. G. Larsen and A. Skou. Bisimulation through Probabilistic Testing. In *Information and Computation 94*, pages 1–28, 1991.
- [LT04] L. Laibinis and E. Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. In *Proc. of the 2nd International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 346–355, 2004.
- [LV62] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal*, pages 200–209, April 1962.
- [Maz09a] Manuel Mazzara. Deriving specifications of dependable systems: toward a method. In *Proceedings of the 12th European Workshop on Dependable Computing (EWDC 2009)*, 2009.
- [Maz09b] Manuel Mazzara. Different perspectives for reasoning about problems and faults. Technical Report CS-TR No. 1151, School of Computing Science, University of Newcastle, April 2009.
- [Maz11] M. Mazzara. On methods for the formal specification of fault tolerant systems. In *DEPEND, International Conference on Dependability*, 2011.
- [MB10a] Issam Maamria and Michael Butler. Mathematical Extension in Event-B through the Rodin Theory Component. Technical report, University of Southampton, June 2010.
- [MB10b] Issam Maamria and Michael Butler. Rewriting and well-definedness within a proof system. In *PAR*, volume 43 of *EPTCS*, pages 49–64, 2010.
- [MB10c] Issam Maamria and Michael Butler. Rewriting and Well-Definedness within a Proof System. In Ana Bove, Ekaterina

- Komendantskaya, and Milad Niqui, editors, *PAR*, volume 43 of *EPTCS*, pages 49–64, 2010.
- [MBER10] Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh. On an Extensible Rule-Based Prover for Event-B. In Frappier et al. [FGK⁺10], page 407.
- [MC08] Christophe Metayer and Mathieu Clabaut. Dir 41 case study. In *ABZ*, page 357, 2008.
- [Meh07] Farhad Mehta. Supporting Proof in a Reactive Development Environment. *International Conference on Software Engineering and Formal Methods*, 0:103–112, 2007.
- [Mes] Message Passing Interface Forum. <http://www.mpi-forum.org/>.
- [MKMA09] Jose Miguel Montanana, Michihiro Koibuchi, Hiroki Matsutani, and Hideharu Amano. Balanced Dimension-Order Routing for k-ary n-cubes. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops, ICPPW '09*, pages 499–506, Washington, DC, USA, 2009. IEEE Computer Society.
- [MM05] A. K. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005.
- [moda] Mode/FT Views wiki page. http://wiki.event-b.org/index.php/Mode/FT_Views.
- [Modb] Modularisation plug-in for the Rodin platform. Documentation at http://wiki.event-b.org/index.php/Modularisation_Plug-in.
- [MOP⁺09] R. Marculescu, U. Ogras, L.-S. Peh, Enright N., and Y. Hoskote. Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.28-1*, pages 3–21, 2009.
- [MPI] MPICH2 website. <http://www.mcs.anl.gov/research/projects/mpich2/>.
- [MRM] MRMC. Markov Reward Model Checker, online at <http://www.mrmc-tool.org/>.

- [MV09] Christophe Métayer and Laurent Voisin. *The Event-B Mathematical Language (Version 2)*, March 2009.
- [Nie01] L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2001.
- [NK04] S. Neema and G. Karsai. Software for Automotive Systems: Model-Integrated Computing. In *ASWSD*, pages 116–136, 2004.
- [NMSSA07] Abbas Nayebi, Sina Meraji, Arash Shamaei, and Hamid Sarbazi-Azad. XMulator: A Listener-Based Integrated Simulation Platform for Interconnection Networks. In *Asia International Conference on Modelling and Simulation*, pages 128–132, 2007.
- [NPW10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. A proof assistant for higher-order logic, 2010. <http://isabelle.in.tum.de/dist/Isabelle/doc/tutorial.pdf>.
- [O’C95] P. D. T. O’Connor. *Practical Reliability Engineering*, 3rd ed. John Wiley & Sons, 1995.
- [OCW03] M. Oliveira, A. Cavalcanti, and J. Woodcock. Arcangel: a tactic language for refinement. *Formal Asp. Comput.*, 15(1):28–47, 2003.
- [OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 1976.
- [O’H07] Peter W. O’Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
- [OL07] Martin Ouimet and Kristina Lundqvist. Modeling the production cell system in the tasm language. Technical report, Massachusetts Institute of Technology, 2007.
- [OMG] OMG. Omg unified modeling language (omg uml), superstructure version 2.2.
- [OMG11] OMG. Requirements interchange format (ReqIF) 1.0.1. 2011.
- [Onl11] Online. Graphical Modeling Framework (GMF). Project Website: <http://www.eclipse.org/modeling/gmf/>, 2011.

- [ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.
- [ORS07] F. Ortmeier, W. Reif, and G. Schellhorn. Formal Safety Analysis of a Radio-Based Railroad Crossing Using Deductive Cause-Consequence Analysis (DCCA). In *EDCC 2005, European Dependable Computing Conference*, pages 139–151. Springer, 2007.
- [OS07] F. Ortmeier and G. Schellhorn. Formal Fault Tree Analysis: Practical Experiences. In *AVoCS 2006, International Workshop on Automated Verification of Critical Systems*, volume 185 of *ENTCS*, pages 139–151. Elsevier, 2007.
- [pat] Transformation patterns plug-in wiki page. http://wiki.event-b.org/index.php/Transformation_patterns.
- [Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge, 1987.
- [PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.
- [PCB98] D. Paun, M. Chechik, and B. Biechelle. Production cell revisited, 1998.
- [PED⁺08] Dongkook Park, Soumya Eachempati, Reetuparna Das, Asit K. Mishra, Yuan Xie, N. Vijaykrishnan, and Chita R. Das. MIRA: A Multi-layered On-Chip Interconnect Router Architecture. *SIGARCH Comput. Archit. News*, 36:251–261, June 2008.
- [Pet11] Petre, L. and Sere, K. and Tsiopoulos, L. and Liljeberg, P. and Plosila, J. Towards Self-Placing Applications on 2D- and 3D-NoCs. In *Autonomic Networking-on-Chip: Bio-inspired Specification, Development, and Verification, Embedded Multi-core System, (EMS) Book Series, CRC Press*, 2011.
- [PFF⁺08] M. Poppleton, B. Fischer, C. Franklin, A. Gondal, C. Snook, and J. Sorge. Towards reuse with “feature-oriented Event-B”. In *Proc. McGPLe: Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, Nashville, TN., October 2008.

- [PHKC09] Maurizio Palesi, Rickard Holsmark, Shashi Kumar, and Vincenzo Catania. Application Specific Routing Algorithms for Networks on Chip. *IEEE Transactions on Parallel and Distributed Systems*, 20:316–330, 2009.
- [PM95a] D. L. Parnas and J. Madey. Functional documentation for computer systems engineering. *Sci. Comput. Program.*, 25:41–61, 1995.
- [PM95b] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995.
- [PMGB05] A. Polychronopoulos, N. Möhler, S. Ghosh, and A. Beutner. System design of a situation adaptive lane keeping support system, the safelane system. In *Advanced Microsystems for Automotive Applications 2005*, pages 169–183. Springer Berlin Heidelberg, 2005.
- [Pop08] M. Poppleton. The composition of Event-B models. In E. Boerger, M. Butler, J. Bowen, and P. Boca, editors, *ABZ2008: Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 209–222. Springer, London, UK, September 2008.
- [PQZ08] Davide Prandi, Paola Quaglia, and Nicola Zannone. *Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, chapter Formal analysis of BPMN via a translation into COWS, pages 249–263. Springer, 2008.
- [Pra03] Praxis. Reveal – a keystone of modern systems engineering. Technical report, 2003.
- [PRI] PRISM. *PRISM probabilistic model checker*. <http://www.prismmodelchecker.org/>.
- [Pro] ProB Model Checker [online]. Available at <http://www.stups.uni-duesseldorf.de/ProB/overview.php> (Accessed 26 October 2011).
- [PS97] J. Plosila and K. Sere. Action Systems in Pipelined Processor Design. In *3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 97)*, 1997.
- [Put05] Martin Putterman. *Markov Decision Processes. Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2005.

- [RB03] Abdolbaghi Rezazadeh and Michael Butler. Event-Based Modelling and Refinement of Distributed Monitoring and Control Systems. In *Refinement of Critical Systems (RCS'03)*, 2003.
- [REB07] Abdolbaghi Rezazadeh, Neil Evans, and Michael Butler. Re-development of an Industrial Case Study Using Event-B and Rodin. In *FACS 2007 Christmas Workshop: Formal Methods in Industry*. BCS London, UK, 2007.
- [RME00] R. Risack, N. Mohler, and W. Enkelmann. A video-based lane keeping assistant. In *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*, pages 356–361, 2000.
- [RODa] RODIN platform website. <http://www.event-b.org>.
- [Rodb] Rodin. The RODIN platform. Online at <http://rodin-b-sharp.sourceforge.net/>.
- [RODe] RODIN Project. at <http://rodin.cs.ncl.ac.uk>.
- [Rod05] Rodin. Rigorous Open Development Environment for Complex Systems (RODIN), 2005. Deliverable D7, Event-B Language, online at <http://rodin.cs.ncl.ac.uk/>.
- [Ros76] B. K. Rosen. Correctness of parallel programs: The church-rosser approach. *Theor. Comput. Sci.*, 1976.
- [Rub95] B. Rubel. Patterns for Generating a Layered Architecture. In J.O. Coplien and D.C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.
- [Rus02] J. Rushby. Using model checking to help discover mode confusion and other automation surprises. In *Reliability Engineering and System Safety*, volume 75, pages 167–177, 2002.
- [Rus09] A.G. Russo. Formal Methods in Industry: The State of Practice of Formal Methods in South America and Far East, September 2009.
- [SB08] C.F. Snook and M.J. Butler. UML-B: A Plug-in for the Event-B Tool Set. In *ABZ*, page 344, 2008.
- [SB09] R. Silva and M. Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In K. Breirman and A.Cavalcanti, editors, *11th International Conference on Formal*

Engineering Methods (ICFEM) 2009, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.

- [Sch11a] Matthias Schmalz. The logic of Event-B, 2011. <ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/698.pdf>.
- [Sch11b] Matthias Schmalz. Term rewriting in logics of partial functions. In *ICFEM 2011*. Springer, 2011. To appear.
- [Sch11c] Matthias Schmalz. Term rewriting in logics of partial functions, 2011. <ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/7xx/732.pdf>.
- [SCZ07] Bill Stoddart, Dominique Cansell, and Frank Zeyda. Modelling and proof analysis of interrupt driven scheduling. In Julliand and Kouchnarenko [JK06], pages 155–170.
- [Sek98] Emil Sekerinski. *Program Development By Refinement*. Springer-Verlag, 1998.
- [Ser90] K. Sere. *Stepwise Derivation of Parallel Algorithms*. PhD thesis, Åbo Akademi University, 1990.
- [SIC] SICS, SICStus Prolog, 2006, website:<http://www.sics.se/sicstus>.
- [Sil] Renato Silva. Parallel composition using Event-B. http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B.
- [Sil11a] R. Silva. Towards the Composition of Specifications in Event-B. In *B 2011*, June 2011.
- [Sil11b] Renato Silva. Renaming framework, 2011. http://wiki.event-b.org/index.php/Refactoring_Framework.
- [SL95] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In *Nordic Journal of Computing*, 2(2), pages 250–273, 1995.
- [SL08] Corinna Spermann and Michael Leuschel. ProB gets nauty: Effective symmetry reduction for B and Z models. In *Proceedings TASE 2008*, pages 15–22. IEEE, June 2008.

- [SPA] SPARKAda. Available at <http://www.praxis-his.com/sparkada/index.asp>.
- [SPB10] Jennifer Sorge, Mike Poppleton, and Michael Butler. A Basis for Feature-oriented Modelling in Event-B. In *ABZ2010*, February 2010.
- [SPHB10] R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 2010.
- [Spi89] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [SPJ08] Colin Snook, Michael Poppleton, and Ian Johnson. Rigorous engineering of product-line requirements: a case study in failure management. *IST*, 50(1-2):112–129, Jan 2008.
- [Sto96] N. Storey. *Safety-critical computer systems*. Addison-Wesley, 1996.
- [Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.
- [SZW05] Jing Sun, Hongyu Zhang, and Hai Wang. Formal semantics and verification for feature modeling. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 303–312, Washington, DC, USA, 2005. IEEE Computer Society.
- [Szy98] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley ACM Press, 1998.
- [TRF03] K. Trivedi, S. Ramani, and R. Fricks. Recent Advances in Modeling Response-Time Distributions in Real-Time Systems. In *Proceedings of the IEEE*, volume 91(7), pages 1023–1037, 2003.
- [TSKA11] Thomas Thum, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof composition for deductive verification of software product lines. *IEEE International Conference on Software Testing Verification and Validation*, 0:270–277, 2011.

- [TTBP02] T.S. Taft, R.A. Tucker, R.L. Brukardt, and E. Ploedereder, editors. *Consolidated Ada reference manual: language and standard libraries*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
- [TTL10] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Towards Probabilistic Modelling in Event-B. In *IFM 2010, Integrated Formal Methods*. Springer-Verlag, 2010.
- [TTL11a] A. Tarasyuk, E. Troubitsyna, and L. Laibinis. Quantitative Verification of Safety in Event-B. In *SERENE 2011, Software Engineering for Resilient Systems*, pages 24–39. Springer-Verlag, 2011.
- [TTL11b] A. Tarasyuk, E. Troubitsyna, and L. Laibinis. Quantitative Verification of System Safety in Event-B. Technical Report 1010, Turku Centre for Computer Science, 2011.
- [TTL11c] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Quantitative Verification of Safety in Event-B. In *SERENE 2011, Software Engineering for Resilient Systems*. Springer-Verlag, 2011.
- [Var09] Kimmo Varpaaniemi. OBSW formal development in Event-B, 2009. Online at <http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html>.
- [vG90] R.J. van Glabbeek. The linear time-branching time spectrum. In *in Proceedings of Theories of Concurrency: Unification and Extension (CONCUR’90)*. Springer, 1990.
- [Vil91] A. Villemeur. *Reliability, Availability, Maintainability and Safety Assessment*. John Wiley & Sons, 1991.
- [Vin93] W. G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*. The Johns Hopkins University Press, Baltimore, 1993.
- [VLH06] M. Verhoef, P. Gorm Larsen, and J. Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In *FM*, pages 147–162, 2006.
- [WAC10] J. Woodcock, E. G Aydal, and R. Chapman. The tokeneer experiments. *Reflections on the Work of CAR Hoare*, page 405–430, 2010.

- [Whi93] D. J. White. *Markov Decision Processes*. John Wiley & Sons,, 1993.
- [Wie03] K. Wiegers. *Software Requirements: Practical Techniques for Gathering and Managing Requirements throughout the Product Development Cycle*. Microsoft Press, Redmond Wash., 2nd ed. edition, 2003.
- [Win90] Jeannette M. Wing. A specifier’s introduction to formal methods. *Computer*, 23(9):8–24, 1990.
- [WKR⁺09] Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proceedings of the 21st IFIP Int. Conference on Testing of Communicating Systems (TESTCOM’09)*, LNCS. Springer, 2009.
- [Wri08] S. Wright. Using Event-B to Create a Virtual Machine Instruction Set Architecture. *Abstract State Machines, B and Z*, pages 265–279, 2008.
- [WRS⁺09] Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, Vitaly Kozyura, Anis Charfi, Frank Michael Kraft, and Ina Schieferdecker. Viewpoints for modeling choreographies in service-oriented architectures. In *Proceedings of the 8th IEEE/IFIP Conference on Software Architecture (WICSA’09)*. IEEE Computer Society, 2009.
- [WRSC08] Sebastian Wieczorek, Andreas Roth, Alin Stefanescu, and Anis Charfi. Precise Steps for Choreography Modeling for SOA Validation and Verification. In *Proceedings of the IEEE 4th International Symposium on Service-Oriented Software Engineering (SOSE’08)*. IEEE Computer Society, 2008.
- [WS98] M. Walden and K. Sere. Reasoning about Action Systems using the B Method. In *Formal methods in System Design*, 13(1), pages 5–35, 1998.
- [YBR10] Sanaz Yeganehfar, Michael Butler, and Abdolbaghi Rezazadeh. Evaluation of a guideline by formal modelling of cruise control system in Event-B. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 182–191. NASA, April 2010.

- [YH10] S. Yoo and M. Harman. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689–701, 2010.
- [ZLT01] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *Tech. Rep.*, 103, 2001.