Design Components

Thesis by

Alexei Iliasov

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

Newcastle University
Newcastle upon Tyne, UK

July 2008

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Alexander Romanovsky, for many insightful conversations and a friendly support during the work on the thesis. This thesis would have never been finished without his help and guidance.

I would like to thank my colleagues from the Abo Akademi, Linas Laibinis and Elena Troubitsyna, from whom I have learnt everything that I know about the B Method. I am particularly grateful to Linas Laibinis for the fruitful collaboration which led to the development of the ideas presented in the thesis.

I am grateful to many people who have helped during the work on the thesis: Maciej Koutny and Cristina Gacek, for spending their time to help me to plan the thesis; Michael Butler, for his criticism of the pattern tool; Jason Steggles, for the idea of pattern correctness verification; Cliff Jones, for reading the early draft of the thesis.

Abstract

Although it is generally recognised that formal modelling is crucial for ensuring the correctness of software systems, some obstacles to its wider adoption in software engineering persist. One of these is that its productivity is low; another that for modelling techniques and tools to be used efficiently, a broad range of specific skills is required. With the gap between computer performance and engineers' productivity growing, there is a need to raise the level of abstraction at which development is carried out and off-load much of the routine work done manually today to computers. Formal modelling has all the characteristics required to replace programming and offer higher productivity. Nonetheless, as a branch of software engineering it has yet to be generally accepted. While there is substantial research accumulated in systems analysis and verification, not much has been done to foster higher productivity and efficiency of modelling activity.

This study puts forward an approach that allows the modeller to encapsulate design ideas and experience in a reusable package. This package, called a *design component*, can be used in different ways. While a design component is generally intended for constructing a new design using an existing one, we base our approach on a refinement technique. The design encapsulated in the design component is injected into a formal development by formally *refining* an abstract model. This process is completely automated: the design component is integrated by a tool, with the corresponding correctness proofs also handled automatically.

To help us construct design components we consider a number of techniques of transforming models and describing reusable designs. We then introduce the concept of model transformation to encapsulate syntactic rewrite rules used to produce new models. To capture high-level design we introduce the pattern language allowing us to build abstraction and refinement patterns from model transformations. Patterns automate the formal development process and reduce the number of proofs. To help the modeller plan and execute refinement steps, we introduce the concept of the modelling pattern. A modelling pattern combines refinement (or abstraction) patterns with modelling guidelines to form a complete design component.

Our approach is both formal and pragmatic. A design component is presented in a consistently formal fashion, which allows it to be analysed and verified. At the same time, it is executable: it can be interpreted and manipulated using software tools.

The thesis is divided into three major parts. The first one discusses model transformations, i.e. simple rules relating formal models. The second part introduces the concept of the pattern as a complex model transformation rule producing an abstraction or a refinement of the input model. The final part develops an approach to guiding the modeller through a development using high-level tactics called modelling patterns. The thesis is concluded with an evaluation chapter illustrating the introduced concepts from the practical viewpoint.

Contents

List of Figures

Chapter 1

Introduction

1.1 Formal Modelling

Finding solutions to complex problems requires careful analysis and validation. Thus, an engineer constructing a bridge uses a mathematical model to predict possible stresses on different bridge parts and to select suitable materials and supporting structures. Not until such analysis has been carefully carried out and thoroughly validated may any construction works start. When constructing a new ship, engineers start with a detailed mathematical model that accurately predicts essential characteristics of the new ship, such as water resistance during movement, maximum load, turn radius and operating costs.

Program construction has so far stood apart. It is not uncommon to build a complex software system by starting directly with some form of implementation and then trying to shape it into a working product through extensive testing. The reason this has worked for many software projects is that the cost of testing and changes can be relatively low for software. There are, nevertheless, several problems with this: exhaustive testing is impossible for most programs; for mission-critical systems the cost of a mistake can be very high; software typically operates in a complex environment which is not always possible to reproduce accurately during testing; in the absence of any analysis or design stage it is impossible to predict how long it may take to construct a satisfactory implementation.

In the context of software engineering, a formal specification is a mathematical description of the system to be constructed. Specification expresses system properties in such a way that it is possible to deduce useful facts about the behaviour of system parts and the system as whole. An important property of a formal specification is that, unlike natural language or semi-formal notations, it is precise and has a formally defined, objective and unique interpretation.

Unlike a program, which provides a concrete implementation of a system, a specification is an abstract design and as such is not executable. This helps to decouple implementation concerns from design decisions. For example, a specification of array sorting is concerned with the properties of a sorted array, while the corresponding implementation deals with the characteristics of the algorithm that constructs a sorted array. Formal modelling presents a number of advantages:

- unambiguous interpretation and documentation;
- an abstract model is much cheaper to construct than a complete implementation, yet the former alone can reveal design flaws and requirements inconsistencies;
- correct-by-construction, formal modelling guarantees that the system behaves correctly provided none of the modelling assumptions is violated during system operation;
- formal verification makes much of the normally required testing redundant.

The benefits of formal modelling come at a cost. It is easier to understand and construct descriptions in a natural language, and many software engineers are not comfortable with the mathematical notation essential to formal modelling. A widespread application of formal methods would therefore require considerable investments into education and training.

The modelling stage delays the moment when there is an executable software version available. There are two reasons for this. Firstly, modelling is not concerned with implementation details and thus executable software appears only as the very last stage of a development. The other reason is the arguably low productivity of formal modelling. The delay, however, is often more than compensated for by a better software quality and correspondingly less time and effort needed for testing.

1.1.1 Classes of Formal Methods

Formal methods rely on differing modelling concepts and principles and thus offer different viewpoints of a system. A formal method is constructed according to a certain paradigm that tries to emphasise particular systems aspects while attempting to hide or play down others. There are several major kinds of formal methods.

A transition-based model describes a system by listing possible transitions. A transition execution is triggered by an event associated with the transition. In most cases, a transition is a state-mapping function which computes a new system state from the given current state. This modelling style is a natural choice for developing reactive systems. In transition model, there is no need to explicitly formulate preconditions on events. A transition precondition is dissolved in the conditions leading to the occurrence of an event which triggers the transition. Event post-conditions are not normally used either which means a specification can be constructed entirely from event-guarded state-mappings. The more notable examples of transition-based modelling methods are Statecharts [1], a formalism which extends finite state diagrams, and PROMELA [2], a specification language for validation of communication protocols.

A history-based model describes all possible execution histories of a system. Such description characterises the system behaviour over a period of time. One important application of this model class is the reasoning about properties of real-time systems, such as absence of race-conditions, deadlock-freeness and liveness. Such properties are normally expressing using temporal logic expressions [3]. Such expressions impose conditions on the past, present and future system behaviour. Likewise, a model of a system specified by formulating all the necessary conditions restricting the model (for example, a system of temporal logic equations) or by explicitly listing all the system histories (e.g., a trace model constructed by animating a CSP [4] process). Pnueli [5] introduced linear temporal logic to reason about properties of concurrent systems. Extensions followed to account for non-linear [6] and continuous [7] time. Lamport [8] developed a modelling method based on a combining a discrete state system and temporal logic.

A process-based model formulates a simple abstract process which is amenable to interpretation and analysis that help to understand the properties of the system under analysis. To verify a system, a modeller can construct an interpretation machine looking for violations of specific properties. Usually this done with a tool called model-checker. Petri-nets [9] and process algebras, such as the CSP [4], CSS [10] and π-calculus [11], are examples of formalisms based on this approach.

A state-based specification describes the state of a system at any given moment. Usually, a state-based model describes a system evolution by means of state-mapping functions. Such functions are defined as relations expressed on the new and old system states but can also have preconditions which describe the states in which applying the function is allowed, and post-conditions which characterise states achievable on applying it [12]. In addition, system states can be constrained with invariants, i.e. properties that must be maintained throughout the system lifetime. State-based specifications may contain redundant elements to allow for interesting consistency verification conditions. Popular state-based formalisms include the Z specification language [13], the Action Systems [14, 15], the B-Method [16] and the Vienna Development Method [17]. The B-Method has recently evolved into a new version called Event-B [18–20].

1.1.2 Practical Formal Methods

For a long time, the focus of formal modelling research was in tackling interesting scientific problems such as constructing novel methods and analysis techniques, while the engineering part received little attention. Formal modelling tool developers were mainly academics not particularly interested in applying their tools to industrial-scale examples. As a result, many such tools suffer from usability and scalability problems, although scalability limitations may be inherent to some approaches. The situation is changing, and the importance of tool support is now universally recognided, with several industrial-strengths tools available and commercially supported [21–25].

Natural languages are very expressive and flexible due to their inherent ambiguity. Such ambiguity, however, may be dangerous when describing a system design and this is why engineers, including software developers, should use restricted, precise and formal languages. Formality makes it possible to conduct objective, impartial design analysis. Texts in formal languages may be harder to write but, unlike natural language texts, they are easier to analyse and manipulate using automated tools.

There are a number of diverse tools and techniques available to assist an engineer constructing a formal model.

A model can be animated to allow a modeller to interactively discover the structure and the underlying algorithms of the model [26]. Animation demonstrates that the formal model matches its informal description and adequately describes the modelled system [27, 28].

A model checker tool can automatically analyse the properties of a formal model. Such tool explores the states or histories of the model, looking for violations of model properties or undesirable situations. Model checking is completely automated and as such requires little effort from a modeller. Input can be just a model itself, and a problem can be reported in the form of a counter-example [29–31]. The availability of powerful model-checking tools has greatly contributed towards the popularity of related formal methods.

An abstract model can be used to generate test cases to confirm the correctness of an implementation derived from the model. This make the testing process more rigorous and possibly cheaper as well [32, 33].

The power of mathematical abstraction makes formal methods applicable to a wide range of problems. Often the interesting properties of a system, even a complex one, can be captured with a succinct abstract model. However, it is not the abstraction of a system that is the final product of a development, but an executable set of instructions, such as a program in a programming language. To retain all the benefits of formal modelling, this must be constructed directly from a model by formally transforming an abstraction into an executable program. The ability to construct a more detailed system model from an abstract model preserving the properties embodied in the abstract model is the cornerstone of formal modelling [16, 34]. A modeller can be offered a toolkit that helping to construct model refinement. For example, refinement calculus defines a set of small-step refinement laws [35]. Specware framework uses category theory to realise powerful model composition strategies and construct refinements from small simple models [36].

1.2 Related Works

1.2.1 Design Patterns

Design patterns enable developers to capture and reuse successful solutions applicable in a range of contexts. The idea of reusable design patterns was originally introduced by an architect Christopher Alexander [37] in mid-seventies. The concept has proved exceptionally successful and over the years it was picked up in other disciplines, including software development. In 1987 Cunningham and Beck [38] proposed an adaptation of the Alexander's pattern language for object-oriented programming. In 1994 Erich Gamma et al. [39] publish a famous collection of design patterns for object-oriented software development. Shortly afterwards the use of these patterns becomes as a fairly standard practice in software development.

Software design patterns, as they are presented by [39], are generic and abstract solutions described in a structured but informal way. Patterns help to communicate important ideas accumulated during the many years of software engineering practice. New patterns can be created along the lines of the existing patterns and complex designs can be communicated by describing them in the terms of widely-known design patterns.

1.2.2 Refinement Calculus

The refinement calculus is a formal framework for constructing executable programs from abstract, possibly non-executable specifications [35, 40, 41]. It is aimed at state-based, imperative style of describing program functionality. The foundational theory is the Dijkstra's weakest precondition semantics [12]. Specifications in refinement calculus are given in the form of predicates linking the previous and the next program states. Such predicate is usually called before-after predicate. A deterministic before-after predicate is a program statement. Before-after predicates come together with preconditions, characterising the valid initial states; and postconditions, describing the desired after-states. The weakest precondition characterises the most general (i.e., weakest) initial state from which application of a given before-after predicates results in states satisfying a given post-condition. Formally, the weakest precondition can be defined as a function wp on two arguments - a before-after predicate S and post-condition P: $wp(S, P)$. According to Dijkstra, the semantics of a statement described by a before-after predicate S is given by value of $wp(S, P)$.

In the refinement calculus, the weakest precondition semantics is extended with the notion of refinement relation between statements (or before-after predicates). The refinement between two statements S and S' is denoted as $S \sqsubseteq S'$ and is defined as follows:

$$\forall p . wp(S, p) \Rightarrow wp(S', p)$$

that is, whenever S establishes a postcondition, so does its refinement counterpart S'. The refinement relation is transitive, antisymmetric and reflexive. The transitivity property makes it possible to organise program construction as a sequence of refinement steps. Thus, to construct a final implementation S_k for an abstraction S, one constructs a sequence $S_1, S_2, \ldots, S_{k-1}$, such that

$$S \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \ldots \sqsubseteq S_{k-1} \sqsubseteq S_k$$

The refinement calculus also defines a set of refinement laws. These are fine-grained refinement steps calculating a refined model version from an abstraction. They are applied in two different ways: to construct a new refinement, applying one law after another; and to verify a refinement instance by demonstrating a chain of refinement laws transforming an abstraction into its refinement.

1.2.3 UML-B

The UML-B approach, proposed by Butler and Snook [42], unites informal, intuition guided modelling of UML with the rigorous modelling approach of the B method (and Event-B for the later versions). Visual UML editor is used as a facade hiding complexities of B models and providing a modelling environment that many software engineers are comfortable with. A tool, called U2B [43], automatically converts a UML model, manipulated by a modeller, into a corresponding B model, which can be analysed with a theorem prover. UML classes are represented as mappings from object instances into individual class properties. Thus, a class with five member variables is modelled using five different functions. Object creation is modelled by adding new mappings to all these functions. A language derived from the B mathematical notation is used to express constrains and actions in UML diagrams.

UML-B is a perfect starting point for translating existing semi-formal designs into formal mathematical models (see, for example [44]). The tool opens the way to reuse of the results of the research on application of design patterns in UML modelling process [45, 46].

The approach has its limitations. The resulting B model is often far from being simple and legible and any attempt at interactive proofs requires careful examination of the generated model. Thus, to do any real modelling a user has to be fairly confident in both UML and Event-B. UML model is always translated into a single Event-B model. While the human eye is very at good dealing with complex structures arranged visually, the textual representation of the same information is much more difficult to comprehend. Since much of the UML underlining ideas come from object-oriented software design, certain concepts are not easily mapped into B. For example, although method is translated as a B event, there is no way to "call" such method from another method in a B specification. Such limitations are inevitable for a combination of two distinct formalism. In the case of UML-B, the Event-B part clearly takes the precedence and the UML concepts are adapted to suit the B modelling style.

1.2.4 B to B0 Generator

Siemens/MATRA, working on specifications of a fully automated train system, have developed a tool for automatically refining B specifications [47]. The tool tries to mechanically produce an implementable model in B0 language (a variant of implementable B) by applying rewrite rules. A large library of such rules were created specifically to handle the specifications of train systems. To make this tool efficient, the use of the B Method is restricted in such a way that the accumulated rule set covers is complete. In other words, for any non-deterministic rule of a model there is a suitable refinement rule. The transformation tool is essentially a code generator with the ability for users to intervene when the tool finds several possible transformations. The bulk of the rewrite rules are concerned with elimination of non-deterministic actions and provision of suitable implementations for abstract data types. The tool was successfully applied in large specification projects (apparently train systems) but is proprietary and is used privately by the company.

1.2.5 Pattern Formalisation and Reuse

There are a number of works investigating the possibility of reuse of design (e.g. in the form of design pattern) in formal developments.

Blazy et al. [48] proposed a mechanism to integrate design patterns into B-Method developments. They also outline guidelines for specifying such design pattern using the B-Method. The pattern instantiation mechanism relies on the model structuring capabilities of the B-Method: inclusion and extensions of machines. A design in this approach is a single B machine. To use it in a development, a modeller has a choice of three different instantiation mechanisms. First mechanism is based on the B `EXTENDS` statement. One or more B machines can be extended by another machine and the operations of the extended become accessible to the parent machine. The idea is that extension of commonly used design solution would help to construct new refinement steps. The second mechanism relies on yet another B structuring statement: `INCLUDES`. Included machines export their variable and invariants but their operations are not visible in further refinement steps. This way, an included pattern is hidden from external observer. In both extension and inclusion instantiation mechanisms, different design patterns are completely disjoint and normally a further effort is needed to integrate them into a development, e.g., by adding new invariants, variables and events. The last instantiation mechanism is simply the conjoining of a pattern with a parent machine. This results in larger models but there are less obstacles in integrating different patterns.

Chan et al. [49] discuss a similar approach but with a emphasise on object-orientation. The work discusses how to model object-oriented development concepts using the B method and, consequently, how to reuse object-oriented design patterns in B. The work present and interesting and practical method for modelling design patterns. The results of such modelling, however, cannot be easily integrated into a development process.

The RAISE Specification Language [50] was used to formalise UML pattern diagrams using a flavour of VDM [17] specification language. A pattern in RSL is a combination of descriptions in a natural language, formally presented pattern structure and a collaboration protocol constraining different parties referenced in the pattern. A pattern is instantiated by first conjoining it with a parent model and then integrating it by renaming its parts.

The LePUS formal framework [51, 52] was developed to formalise the Gang of Four design patterns. Patterns are formalised by expressing their properties as predicates on methods, class properties, class instances and classes. Predicate conjunction is used to construct complex patterns from a collection of simple properties. To our knowledge, the framework has not been used in modelling and software development. This is mainly due to the difficulty of instantiating patterns specified in LePUS.

In [53] Eden et al. describe the design of a tool transforming programming language texts by automatically applying transformations loosely based on design patterns. To use the tool, a programmer first implements a pattern in a special language understood by the tool. Then the tool can use the pattern definition to rewrite a part of a program. Since the tool works at the semantic level of a program, the transformations are done at the level of a programming language structuring units which precludes formulation of high-level, abstract patterns. In addition, not much can be done to ensure correctness of the resulting model or a pattern.

Dwyer at al. [54] proposed to use specification patterns to describe requirements and constraints for models and programs analysed using automated model checkers. Such specification patterns are presented in a way not dissimilar to design patterns [39] and are, essentially, expression templates. An informal pattern description helps a modeller to pick a correct pattern.

The reusability aspect of patterns implies the ability to share and exchange patterns. One possible approach is to describe patterns using some common language that can be connected to a common ontology [55]. Such patterns can be automatically retrieved and matched by tools and an on-line pattern libary can be set-up to foster pattern dissemination [56].

1.2.6 Verifying Compiler

The verifying compiler challenge is a part of the Dependable Systems Evolution Grand Challenge [57]. Such compiler is expected to be able to automatically analyse an input program and prove its correctness. It is understood that such tool will work on a complete program implementations decorated with assertions stating the facts about the intended functionality of the program code. The compiler will try to demonstrate that the assertions are always satisfied and thus the code is indeed safe and correct. This challenge exemplifies the approach when a posteriori analysis is used to demonstrate a system correctness. In this work we follow a different path and rely on a step-wise development procedure that guarantees correctness by construction.

1.3 Motivation and Goals

1.3.1 Guidance during Development

Formal development is essentially a human activity and is known to be time consuming and laborious. To tackle this, a modelling environment must assist a modeller with a constant get feedback on the current development state with an advice on how to proceed next.

There is an extensive body of research on domain-specific software engineering methods. Integration of such methods into the formal development process is essential for constructing large-scale systems.

The larger a system is, the more important it is for us to be able to measure the development progress. Such measure would show how far the development has advanced and what is still left to do.

1.3.2 Design and Modelling Reuse

Formal modelling can only be cost-effective if there is a way to reuse modelling results. A company may be much more willing to invest into a large-scale modelling phase if the solutions discovered at the formal modelling stage can contribute to later related projects.

Just like a large program cannot be created without relying on third-party software, large-scale modelling has to rely on third-party designs. To make this possible, there should be a way to decouple low-level design activity from high-level modelling decisions. Ideally, a company developing a large-scale model of a system should be able to sub-contract or buy the required design parts. Currently, a formal model of a software system is treated as a form of source code and thus has a rather short lifespan. A good design, however, is perhaps more valuable than its concrete implementation. Design reuse could make it commercially viable to create high-quality reusable design products.

1.3.3 Evolution Support

It is hard to obtain a complete and consistent requirements document for a realistic system prior to the design stage. Thus, it is important to be able to adapt formal development to requirements evolution. With their simple semantics and better decoupling of concerns, formal models are bound to be easier to refactor than programming language texts.

1.3.4 Quality-by-Construction

Formal modelling makes it easier to build high-quality products but it does not automatically rule out poor designs. Specification is just a formal translation of an informal system description. The quality of such translation greatly depends on the background and experience of a modeller: no two models constructed according to the very same requirements document would be quite the same. We believe that the ability to extract, package and offer for reuse high-quality formal design procedures will contribute towards the costs and quality of formal developments. This principle is long known in programming - software code libraries are created and maintained by domain experts to be used by mainstream programmers.

1.4 Proposal Overview

In this section we briefly outlay the general rationale behind the thesis. We use the well-known concept of software component to give the intuition on the essence of our proposal: the design component concept.

1.4.1 Software Component

The early history of the computer science is a story of rapid evolution from the programming in low-level machine code to the application of high-level, domain specific programming languages. One of the more important events that shaped the modern software industry was the invention of a software component [58]. A software component can be summarised as a system part providing some predefined services to other components. To implement its services, a component may rely on services provided by other components. An essential property of a component is the hiding of implementation details. An architecture of a component-based system can be understood as an interconnection of a variety of software components.

Unlike a program, a software component is created with reusability in mind. Well-defined interface describes the functionality offered by a component. Inter-

data
Software Component
data

Figure 1.1: A software component is a black box accepting some input and producing some output. Component input and output may be routed to the inputs and outputs of other components.

face of a component is typically general enough to allow developers to use the component in different contexts and different system types. Software component is a black-box - it accepts some input and produces some output without exposing details of internal computations (Figure 1.1). Normally, a programmer cannot look inside and change the component logic. This done not only to conceal implementation details but also to ensure that an overall system can be understood as a composition of components with known properties. The quality of components constituting a system may serve as an indication of the overall system quality.

The same component interface can be implemented by different components in different ways. Implementation diversity is important for a number of reasons: a component implementation may evolve without violating interface to other components, an abstract interface may permit a large number of different implementations; each adapted to particular conditions and requirements; the ability to replace a component for a different component with the same interface facilitates healthy competition in the software market.

1.4.2 Design Component

Mathematical modelling has the powerful tool of abstraction: it is always possible to find a level of abstraction at which a system description is simple enough to be amenable to formal analysis and yet comprehensive enough to states interesting facts about the system. To construct the whole system and to ensure that result is error-free, an abstraction is gradually developed up to the stage when a model becomes a program and can be unambiguously interpreted by a computer. Such detalisation is a difficult and laborious process. It achieves everything a programming would achieve but in the process a developer also constructs the proofs of the design correctness and presents many intermediate steps that can be used for mathematical analysis of the system properties and validation of the system design.

In this work we investigate a mechanism which introduces structuring and reuse into formal developments. By analogy with the concept of software component, we call this structuring unit a *design component*. The ultimate ambition of the design component mechanism is to bring the advantages of componentrised development into formal modelling.

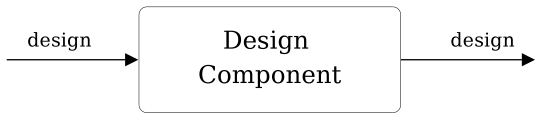


Figure 1.2: A design component accepts a design and produces a new design. Input and output may be routed to other design components.

A design component accepts some design (which may be called a specification or a model) and produces a new design (Figure 1.2). Like a software component, a design component is reusable for a class of problem domains. Design components can be composed together - an output of a component can be routed to the input of another component. Design component is also a black box - a modeller can only investigate a component interface but the examination of component internal workings is not needed to apply the component.

1.5 Problem Statement

The process of construction of large-scale formal models has not been widely researched so far. Formal modelling experts are more interested in identifying and solving fundamental problems in the area of modelling and analysis of information systems. Industrial uses perceive formal modelling as too difficult and expensive and prefer to rely on semi-formal techniques, such as UML [59], or completely automated tools [60], that can be used by untrained programmers.

We would like to be able to present a modeller with a collection of modelling strategies - problem-specific guidelines on model construction - and automated model transformations. We understand a modelling strategy as an active model assistant guiding a modeller through a development. Unlike a paper-based method, such modelling strategy would be interpretable by a machine and a modeller would be presented the results of the strategy interpretation in the context of a current development. We believe that many existing domain-specific software engineering methods can be converted into modelling strategies.

To make the modelling process easier, we should be able to automate parts of it. Formal models of large system are bound to reuse many well-known design ideas, such design and architectural patterns (e.g., Triple-Modular Redundancy), and information processing solutions (e.g., a sorting algorithm, a communication buffer model). Despite being well-known, with the current state-of-the-art, they still have to be redesigned every time a new. This is a waste of time and an indication of a poor organisation of the modelling process. Once a design is constructed it should be reusable in different developments.

The necessity to conduct a substantial number of interactive proofs is one of the major impediments to the.wide adoption proof-based verification methods. The possibility of combining proof reuse with design reuse would make formal modelling more attractive and accessible to non-experts.

We believe the problems we consider are highly relevant to the formal methods and software engineering communities. The *Roadmap for Enhanced Languages and Methods to Aid Verification* [61] lists refinement patterns as one of the long-term research goals. Another long-term goal is *Evolution + Refinement*, which argues for the need to support development restructuring. We demonstrate in the thesis that these two goals are closely related and can be addressed with a common solution.

1.6 Thesis Overview

In the thesis we introduce several design reuse mechanisms, ranging from simple model rewrite rules to high-level modelling tactics. These are used to build complex design reuse units which are amenable to mathematical analysis and automatic interpretation and application. Such units are used to assemble design components - the ultimate goal of our research - that assist a modeller in construction of formal developments.

The first of the design reuse mechanisms, called model transformation, describes simple model rewrite rules. The principal application of model transformations is the construction of a library of basic transformations that can be used to change formal models. In addition, we are able to define model transformations that abstract from the peculiarities of a given formalism and address a whole family of related formalisms. Chapter 3 is devoted to the discussion of model transformations. It introduces model transformations for the Event-B method (we overview the Event-B method in the next chapter).

We continue with the discussion of abstraction and refinement patterns. We introduce the pattern language to describe complex model transformations. The language is independent of any formalism; the language constructs compose formalism-specific model transformations to describe how one model is transformed into another model. An abstraction pattern always constructs an abstraction of a given concrete model whereas a refinement pattern always delivers a refinement of a given abstract model. We use a proof theory to generate proof obligations that would statically demonstrate that an abstraction pattern indeed succeeds in construction of a model abstraction and application of a refinement pattern always yields a correct model refinement. The pattern language and the related topics are discussed in Chapter 4. We illustrate the use of the patterns mechanism with the Recovery Block refinement pattern, based on the Recovery Block mechanism [62].

We believe that it is not enough to provide mechanised refinement steps. A modeller should be given an advice when to apply a specific refinement pattern and how to instantiate it. More importantly, at any given point during a development process, a modeller should be able to get an advice from a tool on how to proceed next with the development. We introduce the concept of a modelling pattern - a high-level modelling strategy describing a succession of steps leading to the satisfaction of development goal. Some such steps are handled by refinement or abstraction patterns while others simply restrict the way a manual refinement step is conducted. Modelling patterns are discussed in Chapter 5. We use a code generation case study to illustrate the modelling pattern development process.

The evaluation chapter6 discusses several examples of refinement patterns and our tool prototype for the RODIN Event-B Platform [63].

Finally, in Chapter 7, we give a high-level overview of our approach and compare it to some related works.

Chapter 2

Background

2.1 Event-B Method

This section presents an overview of the Event-B method [20]: the syntax and the structure of an Event-B machine, well-formedness conditions and refinement proof obligations.

An Event-B development is a collection of models. Each such model is represented using programming language-like notation called Abstract Machine. An abstract machine has some local state, characterised by its variables and a number of state updating operations. In Event-B such operations are called events. Unlike programming language procedures or classical B [16] operations, Event-B events cannot be invoked explicitly by some other part of a model. An event may be executed only when its guard (a form of a precondition) is enabled.

An Event-B development is a chain of Event-B models. The first model in a development is called an abstract machine. An abstract machine defines some local variables and provides events for the state evolution. An abstract machine has the following general form:

SYSTEM $SendRecv$
SEES $context$
VARIABLES v
INVARIANT I
INITIALISATION R_I
EVENTS
  $e_1 \quad = \quad \ldots$
  $\ldots$
  $e_n \quad = \quad \ldots$

Note the distinguished event which provides initial states for model variables.

A machine declaration starts with clause define a machine name. Carrier sets and constants are defined in separate modules, called contexts. Contexts can be made visible to a machine but are otherwise independent (the same context can be seen by unrelated models). Model variables is simply a list of variable names. Typing predicates for variables are incorporated into a model invariant. An invariant may also define additional constraints on the reachable model states. The initialisation event contains action initialising model variables.

Model events are atomic. Once a model event starts execution no other event may start. An event with a guard G and a collection of actions R has the following syntax

$$
\begin{array}{lll}
e & = & \textbf{WHEN} \\
 & & \quad G \\
 & & \textbf{THEN} \\
 & & \quad R \\
 & & \textbf{END}
\end{array}
$$

Often it is convenient to use local variables (parameters) to compute a new model state. Such extended form of an event has the following syntax

$$
\begin{array}{lll}
e & = & \textbf{ANY}\ p\ \textbf{WHERE} \\
 & & \quad G \\
 & & \textbf{THEN} \\
 & & \quad R \\
 & & \textbf{END}
\end{array}
$$

where p are the event parameters. Parameters play dual role. They can be used to simplify event actions by splitting computation of an state into two stages: the assignment to local variables and then the assignment to machine variables. The other role is to use parameters to distinguish between different variants of the same event. For example, an event removing an arbitrary item from a set can be declared like this

$$
\begin{array}{lll}
e & = & \textbf{WHEN} \\
 & & \quad set \neq \varnothing \wedge finite(set) \\
 & & \textbf{THEN} \\
 & & \quad set :| card(set') = card(set) - 1 \\
 & & \textbf{END}
\end{array}
$$

or, using a local variable, like this

$$\begin{array}{rcl} e & = & \textbf{ANY } i \textbf{ WHERE} \\ & & \quad i \in set \\ & & \textbf{THEN} \\ & & \quad set := set \setminus \{i\} \\ & & \textbf{END} \end{array}$$

The use of local variable makes the event declaration more readable and has the additional benefit of of identifying the item removed when animating the model. Event-B uses the theory of generalised substitutions [64] to describe how an action transforms a model state. The table below lists the substitution styles that are used to describe an action

notation	$relationpredicate$	
$v := F(c, s, v, l)$	$v' = F(c, s, v, l)$	assignment
skip	$v' = v$	no-effect assignment
$v :\in F(c, s, v, l)$	$v' \in F(c, s, v, l)$	set choice
$V :\mid F(c, s, V_0, V_1, l)$	$F(c, s, V_0, V_1, l)$	generalised substitution

where v is a variable, F is an expression, V is a vector of variables and V_0 and V_1 are the new and old values of V. Expression F may refer constants c, sets s, system variables v and local variables l.

The first of substitution type - := - is a simple assignment. The assigned variable becomes equal to the value of expression F. Substitution $v :\in F$ selects a new value for v such that it belongs to set F. The most general substitution operator $:\mid$ uses a predicate to link the new and old model states. Several substitutions (actions) can be combined into a complex action using the parallel composition operator

$$s_1 \| s_2 \| ... \| s_k$$

2.1.1 Consistency Checking

An Event-B model is specified in such manner that it contains redundant information which helps to verify that different model parts agree with each other. In particular, consistency checking is concerned with demonstrating that each model event established the model invariant provided it is invoked in a state satisfying the invariant and the event guard. This condition is expressed as follows:

$$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \wedge R(s, c, v, v') \Rightarrow I(s, c, v')$$

where s and c are sets and constants imported from contexts, v and v' are variables describing the previous and the new model states, P are the known constraints on constants and sets, I is the model invariant, G is the guard of the current event and R is a before-after predicate computed from the event actions.

An action of an event must be able to compute a new system state once it is enabled by the event guard. This is expressed as the feasibility condition and defined as follows:

$$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \Rightarrow \exists v' \cdot R(s,c,v,v')$$

These two rules take different forms for different styles of event declaration [20]. An important special case is the initialisation event which. This event cannot assume any previous state and thus does not have a guard. The corresponding condition are

$$P(s,c) \wedge R_I(s,c,v') \Rightarrow I(s,c,v')$$

$$P(s,c) \Rightarrow \exists v' \cdot R_I(s,c,v')$$

Also note the absence of v in R_I initialisation before-after predicate: an action computing an initial variable state has no previous state to refer to.

Sometimes it is important to ensure that a model cannot deadlock. This can be done by stating that the disjunction of all the guard of all the model events is never false, that is, there is always at least one enabled event

$$P(s,c) \wedge I(s,c,v) \Rightarrow \bigwedge_{i \in 1..n} G_i(s,c,v)$$

2.1.2 Event-B Refinement

Event-B is based on step-wise development which detalises a system through a number of correctness preserving steps, called refinements. The refinement process with the creating of an abstract model and finishes with construction of a detailed and deterministic from which an executable code can be generated. The refinement process aims to reduce nondeterminism of the abstract specification and replace abstract mathematical data structures by data structures implementable on a computer, and, hence, gradually introduce implementation decisions.

The structure of an Event-B refinement machine is essentially identical to the structure of an abstract machine. The only addition is the clause pointing at the abstraction of a current model

$\cdot \mapsto \cdot$	$a \mapsto b$	$\{1 \mapsto 2\}(1) = 2$	mapping or a pair
$\cdot \lhd \cdot$	$d \lhd f$	$\{1\} \lhd F = \{1 \mapsto 2\}$	domain restriction
$\cdot ⩤ \cdot$	$d ⩤ f$	$\{1\} ⩤ F = \{2 \mapsto 3\}$	domain subtraction
$\cdot \rhd \cdot$	$f \rhd r$	$F \rhd \{2\} = \{1 \mapsto 2\}$	range restriction
$\cdot ⩥ \cdot$	$f ⩥ r$	$F ⩥ \{2\} = \{2 \mapsto 3\}$	range subtraction
$\cdot[\cdot]$	$f[s]$	$F[1..2] = \{2, 3\}$	image of a relation or a function
$\cdot^{-1}$	f^{-1}	$F^{-1}(3) = \{2\}$	inverse of a relation or a function
$\cdot ; \cdot$	$f ; g$	$F ; F^{-1}(2) = 2$	relational composition
$\cdot \mathbin{<\!\!+} \cdot$	$f \mathbin{<\!\!+} g$	$F \mathbin{<\!\!+} F^{-1}(2) = 3$	relational override
$\cdot \times \cdot$	$A \times B$	$\mathbb{Z} \times \{1\}$	cartesian product
$\cdot \rightarrow \cdot$	$A \rightarrow B$	$\mathbb{Z} \rightarrow \mathbb{N}$	total function
$\cdot ⇸ \cdot$	$A ⇸ B$		partial function
$\cdot ⤔ \cdot$	$A ⤔ B$		partial injection
$\cdot \rightarrowtail \cdot$	$A \rightarrowtail B$		total injection
$\cdot ⤀ \cdot$	$A ⤀ B$		partial surjection
$\cdot \twoheadrightarrow \cdot$	$A \twoheadrightarrow B$		total surjection
$\cdot ⤖ \cdot$	$A ⤖ B$		bijection

where $F = \{1 \mapsto 2, 2 \mapsto 3\}$

Figure 2.1: An overview of the mathematical notation specific to the Event-B method.

REFINEMENT m_1
REFINES m
SEES $context$
VARIABLES u
INVARIANT J
INITIALISATION S_I
EVENTS
$e_1 \;=\; \ldots$
$\ldots$
$e_n \;=\; \ldots$

The refinement relation between two Event-B models is demonstrated by checking a number of conditions on the events of the concrete model. The first such conditions requires for an event inherited from the abstract machine that its action is a correct refinement of the abstract action

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,u) \wedge H(s,c,u) \wedge S(s,c,u,u') \Rightarrow v' \cdot (R(s,c,v,v') \wedge J(s,c,v',u'))$$

where J is the invariant of the concrete model. This invariant constraints concrete model variables u and may also refer to the abstract model variables v. The part of the invariant linking abstract and concrete variables is called a gluing invariant. H is the concrete event guard and S is the concrete before-after predicate.

The feasibility condition requires that a concrete action is computable when both abstract and concrete invariants and the event guard are enabled

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,u) \wedge H(s,c,u) \Rightarrow u' \cdot S(s,c,u,u')$$

The guard of the concrete version of an event must be stronger that its abstract counterpart

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,u) \wedge H(s,c,u) \Rightarrow G(s,c,v)$$

Model refinements may also introduce new events. New event of a model is allowed to modify only the new variables, introduced in the concrete model. For such event the refinement conditions are as follows:

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,u) \wedge H(s,c,u) \wedge S(s,c,u,u') \Rightarrow J(s,c,v,u')$$

and

$$P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w) \Rightarrow u' \cdot S(s,c,u,u')$$

Event-B mathematical language is mostly within the boundaries of the standard set-theoretical notation. Event-B defines few extentions and fixes the syntax for some operators that do not have a single widely used notation 2.1.

2.2 Goal-Oriented Requirements Engineering

A high-quality requirement document is a prerequisite to construction of an adequate formal model and, consequently, executable implementation. Formulation of requirements is far from trivial exercise. The use of inconsistent, incomplete and ambiguous requirements is bound to result in poor quality software. The requirements collection and analysis stage helps developers to distinguish between the requirements to a system to be realised and assumptions about the operational environment of the system. A late discovery of a problem with requirements would often result in costly changes to a current faulty implementation and often the whole development has to be abandoned as the result [65].

The importance of requirements engineering and analysis stage has been recognised as soon as software engineering began prevailing over hardware engineering

[66]. As a result, a number of requirements engineering methods emerged to assist a developer in collecting, structuring and validating requirements. On of the more notable examples of such methods is the goal-oriented requirements engineering [67, 68].

Goal-oriented requirements engineering served as an inspiration for an approach to formulate reusable development strategies (modelling patterns). Further in this section we briefly present the foundational principles of goal-oriented requirements engineering.

In goal-oriented requirements engineering, a goal is understood as an objective to be achieved during a system development. The initial set of goals covers high-level system properties. Such goals are formulated from the viewpoint of a prospective system user. A high-level goal is the departure point for identifying more technical, low-level goals and also more general, architectural goals. The transition to lower-level goals is called goal refinement [67]. With the goal refinement, an abstract goal is mapped into a set of new goals that help to understand *how* the abstract goal in question is to be reached. A goal is abstracted by finding a more general goal that explains *why* a given is to be achieved. Ultimately, the goal abstraction process leads to a single goal from which all other goals are derived through the refinement process. Refinement and abstraction are the two main goal elaboration mechanism. They can be applied with a varying degree of formality depending upon the style of goal representation. Formal constraints on goal help to identify inconsistencies in requirements. Through a formal analysis it is possible to discover missing goals, and also identify and resolve goal conflicts.

Prior to a formal analysis, one has to find a way to formally represent goals. The approach formulated in [69] advocates formalisation of a goal as a predicate expressing the goal satisfaction conditions in the terms of constraints on objects relevant to the analysed system. Formal goals open many opportunities, most importantly it makes possible to off-load some goal analysis to an automated tool. The predicate-based formalisation approach has its limitations. Some goal kinds, for example, safety and performance goals, do not lend themselves to a tractable presentation in the predicate calculus. One solution is to clearly separate the requirements of different kinds and apply specific formalism to each requirement kind.

In this work, when we come to the discussion of modelling patterns, we show that a goal can be understood as a meta-model - a description of a class of models satisfying a given criterion. With this viewpoint the refinement-based development fits naturally the idea of satisfying a goal during model construction. Individual requirements are linked to form a path that a modeller follows working on a model development.

Writing out goals, a modeller uses a special notation. Besides a name, a goal can be decorated with attributes, such as priority (qualitative or quantitative) and perceived goal feasibility. The abstraction and refinement relation between goals are fundamental to requirements analysis and elaboration. Goal links are used to express the relation between goals. The *and-refinement* link connects a goal to a set of sub-goals and satisfaction of all the sub-goals automatically implies the satisfaction of the parent goal. The *or-refinement* link is used to demonstrate that a goal can be satisfied by satisfying either of its sub-goal. Sometimes, an exclusive or-refinement is used to explicitly that sub-goals are conflicting and only one can be satisfied. A more general *conflict* can be used to relate mutually exclusive goal. One of such goal must be eliminated to produce the final requirements. Conflict links are often used to describe product line requirements where they play the role of a variation point. The two kinds of refinement links define the two main goal refinement techniques - *and* decomposition and *or* decomposition. Specific methods built around the goal-orientation concept introduce additional types of links. The i^* framework [70] assigns goal to agents and additional dependency links can be used to model cases when an agent depends on another agents to fulfil a goal. KAOS [67, 71] permits operationalisation links mapping goals into operations of the implemented system.

Chapter 3

Model Transformation

3.1 Introduction

For anything but a toy example it is hard to construct a formal model without the step-wise development procedure (here we understand step-wise refinement as it is introduced in [72]). Refinement helps to structure a large development into a succession of design decisions, each concerned with a particular aspect of a system functionality. Although, in a general case, a refined model may have very little in common with its abstraction, in practice, this does not happen often as such refinement steps are hard to construct, understand and verify. A refinement introducing many changes requires a high level of confidence in the introduced design decision and thus is costly to change or discard. This is why smaller refinement steps should be preferred.

Small refinement steps come naturally in the form of the transformational approach to the refinement. Instead of writing a new specification from a scratch and then proving the refinement relation, a new refinement can be produced by modifying a small model part, leaving most of the model intact.

Model transformations are conveniently described as model rewriting rules. A model transformation produces a new model by rewriting a model part according to a simple rule. Some transformations may accept parameters which are provided externally during the application of a model transformation.

In this chapter we discuss the concept of model transformation and define a collection of model transformations for the Event-B method. We show how to compose transformations to construct a complex transformation and how to describe refinement steps with a composition of model transformations.

3.2 Model Transformation

A formal model is normally understood as a mathematical description of a system and is interpreted according to the semantics of the applied method. Model transformation is the process of obtaining one model from another. It can be described as a mapping of a model into another model:

$$T : M \rightarrow M$$

where M is a universe of models and T is a model transformation. A concrete model is obtained by applying some T to an abstract model:

$$m' = T(m)$$

This definition requires that a model transformation accepts any model. It is more convenient to have T defined as a partial function:

$$T : M \nrightarrow M$$

Now, a model can be transformed only when it is in the domain of the transformation:

$$m \in dom(T) \Rightarrow m' = T(m)$$

An important class of model transformations are transformations that always construct a refinement of an input model. To show that a model transformation constructs a refinement, one has to demonstrate that for any acceptable input abstract model, the result produced by the transformation rule is a valid refinement:

$$m \in dom(T) \land m' = T(m) \Rightarrow m \sqsubseteq m'$$

A practical way to define the domain of a model transformation is to provide a characteristic function, called *model template*. Such template - a predicate on model properties - describes a class of models:

$$M_\mu = \{m \mid \mu(m)\}$$

where M_μ is a set of models satisfying template μ. An input model acceptable by a transformation is one satisfying the template predicate of the transformation. There are two important classes of models: an input model class and an output model class. The input model class of a transformation characterises model accepted by the transformation while the output model class describes the possible the results of a transformation. These two classes are defined as follows:

$$dom(T) = \{m \mid \mathbf{incl}(T)(m)\}$$

$$ran(T) = \{m \mid \mathbf{outcl}(T)(m)\}$$

A model transformation may be parametrized. Instead of accepting just an input model, a transformation can also accept a vector of parameters. This can be described as a family of model transformations:

$$T : P \nrightarrow M \nrightarrow M$$

The following syntax is used to specify model transformations:

$$\begin{array}{lcl} \mathsf{name}(p,s) & \equiv & \textsf{requirements} \\ & & \quad c(s,p) \\ & & \textsf{effect} \\ & & \quad s' = r(s,p) \\ & & \textsf{ref} \\ & & \quad ref \end{array}$$

where p is a vector of parameters, s is an input model, $c(s,p)$ is an applicability condition and $r(s,p)$ is a rule computing the refined version s' of an abstract model s. The $ref \in BOOL$ flag states whether a given model transformation produces a valid model refinement (abstraction). This is a constant property of a model transformation and the way it is computed is dictated by the refinement/abstraction conditions of a specific method. A modeller instantiates a model transformation by supplying parameters and an abstract model to be transformed. The result, a new model s', is constructed by computing the model transformation rule $r(s,p)$. The applicability condition of a model transformation also defines the input model class of the transformation, thus $c(s,p) = \mathbf{incl}(name)(s,p)$ where $name$ is the name of the transformation. The output model class is not specified explicitly but is easy to derive from the rule $r(s,p)$ computing a new model:

$$\mathbf{outcl}(name) = c(s,p)[r(s,p)/s]$$

To summarise, a model transformation is rule producing a new model by transforming some input model. A model transformation is associated with two model classes: the input model class that describes the set of all models accepted by the transformation and the output model, characterising the set of possible transformation results. The input and output models of a model transformation must belong

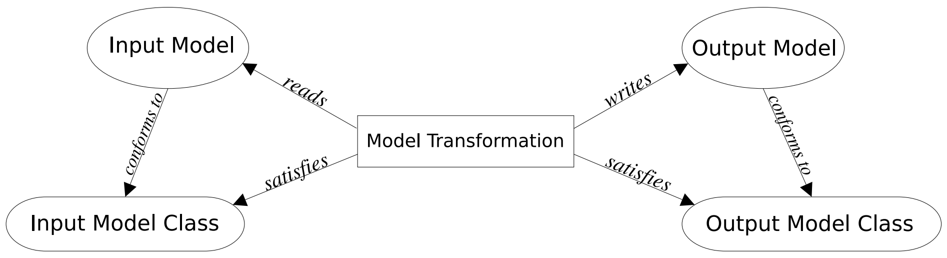


Figure 3.1: The relations between the major concepts of model transformations.

to, correspondingly, the input and output model classes (Figure 3.1).

3.3 Inverse Transformation

It is interesting to see if the effect of a model transformation can be undone by another model transformation. In particular, for a given model transformation md we want to be able to compute its inverse md^{-1}:

$$md \circ md^{-1} = \mathbf{id}_M$$

where md^{-1} is both the left and right inverse of md. Unfortunately, in the general case, model transformations are not injective and an inverse form cannot be computed for an arbitrary transformation. To construct inverse transformation we first have to extract the class of injective (information-preserving) transformations. Such transformations add new model elements without removing or changing abstract models elements and thus can be unambiguously undone:

$$T_a : P \leftrightarrow M_i \rightarrowtail\!\!\!\!\twoheadrightarrow M_o, \text{ where } M_i \subseteq M \wedge M_o \subseteq M$$

where T_a is a part of T with the doamin restricted to inversible model transformations: $T_a \subseteq T$. The inverse of a model transformation is then an inverse of model mappings:

$$T_a^{-1} : P \leftrightarrow M \leftrightarrow M \wedge \forall p \cdot (p \in P \Rightarrow T_a(p) \circ T_a^{-1}(p) = \mathbf{id}_M)$$

For a model transformation

$$\begin{array}{lcl} \mathsf{n}(p) & \equiv & \textsf{requirements} \\ & & \quad c(s,p) \\ & & \textsf{effect} \\ & & \quad s' = r(s,p) \\ & & \textsf{ref} \\ & & \quad r \end{array}$$

the inverse transformation is defined as

$$\mathsf{n}^{-1}(p) \equiv \begin{array}{l} \textsf{requirements} \\ \quad c(s,p)[s_0/s] \wedge s = r(s_0,p) \\ \textsf{effect} \\ \quad s' = r^{-1}(s,p) \\ \textsf{ref} \\ \quad r \end{array}$$

provided that inverse form of r, r^{-1} exists.

3.3.1 Composing Model Transformations

A method equipped with a set of model transformations is a step towards guided model construction. With a palette of transformations, model refinements can be constructed as series of model transformation steps.

To conduct complex refinement steps with a set of predefined model transformations we need a way to combine several model transformations into a complex one which would produce the desired refinement step. With the sequential composition, model transformations are applied one after another

$$t_1; t_2; \ldots; t_n$$

The sequential composition of two model transformations results a new model transformation. For two generic model transformations -

$$\mathsf{t}_1(p_1) \equiv \begin{array}{l} \textsf{requirements} \\ \quad g_1(p_1,m) \\ \textsf{effect} \\ \quad m' = a_1(p_1,m) \\ \textsf{ref} \\ \quad r_1 \end{array}$$

and

$$\mathsf{t}_2(p_2) \equiv \begin{array}{l} \textsf{requirements} \\ \quad g_2(p_2,m) \\ \textsf{effect} \\ \quad m' = a_2(p_2,m) \\ \textsf{ref} \\ \quad r_2 \end{array}$$

- the sequential composition is computed as follows:

$$
\begin{array}{rcl}
\mathsf{t}_1; \mathsf{t}_2(p_1, p_2) & \equiv & \textsf{requirements} \\
& & \quad g_1(p_1, m) \wedge m_0 = a_1(p_1, m) \wedge g_2(p_2, m_0) \\
& & \textsf{effect} \\
& & \quad m' = a_2(p_2, m_0) \\
& & \textsf{ref} \\
& & \quad r_1 \wedge r_2
\end{array}
$$

where $g_1(p_1, m) \wedge m_0 = a_1(p_1, m) \wedge g_2(p_2, m_0)$ is the well-formedness condition for the composition of transformations: transformations may be composed only if the result of the first one ($a_1(p_1, m)$) belongs to the class of input models accepted by the second one ($g_2(p_2, m_0)$). Rule $r_1 \wedge r_2$ states that a composition of refinement (abstraction) model transformations results in a new refinement (abstraction) model transformation whereas a composition of non-refinement or a refinement and a non-refinement is not automatically a refinement model transformation.

The above also defines the rule for computing an input model class for a composition of model transformations. Inverse of a composition is the reversed composition of the inverse transformations:

$$(t_1; t_2 \ldots t_n)^{-1} = t_n^{-1}; t_{n-1}^{-1}; \ldots; t_1^{-1}$$

provided that inverse forms t_i^{-1}, $i \in 1..n$ exist. This rule is a direct consequqnce of the inverse of a function composition.

3.4 Model Annotations

Formal models, strictly speaking, are not completely formal. A formal model contains such informal elements as names, comments and literal constants. These are not given any interpretation in a formal method although the information contained in those elements is essential to understanding a model. An obfuscated model, just like an obfuscated program, is virtually useless. The description of a model purpose, its application domain and the expected behaviour are normally done in a natural language. Thus, to completely validate a model, one also has to consider such informal descriptions. Verification tools are only concerned with the part of a model semantics relevant to verification. Automated model transformation, however, is one of the cases when a software tool is interested in knowing model purpose, problem domain and the roles of its parts. This information helps to reason about transformation in the terms of the model problem problem, identify a class of suitable transformations, and rule out the class of transformations that should not be applied.

We cannot hope to be able to interpret annotations written in a natural language and thus have to restrict ourselves to some form of a machine-readable notation. One way to do this to use a formal ontology [73]. An ontology provides an objective and unambiguous description of phenomena from the problem domain of a modelled system.

Our proposal is to extend model with *annotations*. An annotation is a formal text written using some (shared) ontology. Such annotations have two main applications: they provide additional information about model elements and, thus, help to construct more elaborate patterns; they describe the purpose and the application domain of a transformation.

A model annotation is also used to characterise a model part. A variable can be said to play a role of a temperature sensor or a program counter or an actuator. Information like this cannot be deduced from a model alone. Such high-level, domain-specific descriptions may be essential to construct interesting model transformations and to allow a modeller to reason about development parts in domain-specific terms, as opposed to manipulating low-level model elements.

It is important to give annotations in a context. Annotation characterising, for example, an event of model is attached directly to that event. A model element can have any number of annotations. The simplest form of annotation is an attribute attached to a model element. For example, a variable recording pressure can have the "barometer" attribute. This attribute helps to realise the role of the variable in a model in cases when the information contained in a model itself is not enough. To avoid possible mismatch of measurement units, the variable may also carry an additional attribute stating the fact that the pressure is measured in Pascals. This can be expressed by attaching a new annotation containing pair "unit=pascal".

More complex annotations can be constructed using a knowledge representation language, such as KIF [74]. In KIF, one can construct layered descriptions, going from very low-level statements about model elements up to the high-level properties, expressed in terms of the problem domain of a modelled system. For example, an aircraft control system could be described as a composition of an altitude and landing controls. The altitude control may comprise measurement, prediction and actuation components. With a knowledge representation language we are able to map the concepts characterising various system components into model annotations and syntactic constrains.

Knowledge representation languages, such as KIF, are expressive and flexible. In addition to a simple taxonomy, an ontology can also include relations and predicates. For example, the *updates* relation, given below in the KIF syntax, states that a given event updates some given variable

```
(forall (?V ?E) (=> (and
  (instance ?V variable) (instance ?E event)
  (exists ?A (and (instance ?A action)
    (member ?A ?E) (member ?V ?A))))
  (updates ?E ?V)))
```

This expressions defines a new predicate $updates$. It assumed that classes $event$, $action$, $variable$ are defined elseweher. The rest elements are the standard features of the KIF language.

Formal ontology and knowledge representation are supported by a number of tools and open databases available for reuse and integration. For example, there is a free implementation of the KIF language [75] which can be used to enhance our design components tool.

We assume that any syntactic model element can be decorated with an unlimited number of annotations. Formally, this is expressed by adding function $annot$ to all the elements of a model. Then the following two model transformations can be defined. Model transformation annotate adds a new annotation with a given tag to a syntactic unit of a model

$$\begin{aligned} \mathsf{annotate}(e, t, a) \quad \equiv \quad & \textsf{requirements} \\ & \quad true \\ & \textsf{effect} \\ & \quad annot'_e = annot_e \mathbin{\lhd\!\!\!+} \{t \mapsto a\} \\ & \textsf{ref} \\ & \quad true \end{aligned}$$

Transformation deannotate removes an existing annotation or does nothing if there is no matching annotation

$$\begin{aligned} \mathsf{deannotate}(e, t) \quad \equiv \quad & \textsf{requirements} \\ & \quad true \\ & \textsf{effect} \\ & \quad annot'_e = \{t\} \mathbin{\lhd\!\!\!-} annot_e \\ & \textsf{ref} \\ & \quad true \end{aligned}$$

These model transformations are formulated without any assumptions about the structure and semantics of a modelling method and thus they are method-neutral. We could even formulate some method-independent patterns using these transformations.

3.5 Event-B Model Transformations

This section defines a collection of model transformations for the Event-B method. First we describe the structure of Event-B models and the way to represent them as a syntactic tree.

At the top level, an Event-B model is characterised by its name, a link to a parent model, a set of variables, an invariant and a set of events.

SYSTEM nom (machine name)
REFINES par (machine parent)
VARIABLES
  var (set of variables)
INVARIANT
  inv (set of model invariants)
INITIALISATION
  (variable initialisations)
EVENTS
  evt (set of model events)

We ignore name and parent parts as they do not play any role in a model transformation description and are only needed to define the ordering on models of a development. An Event-B model is described by a tuple of variables, invariants and events (also Figure 3.2):

$var \subset \mathrm{VAR}$	set of variables
$inv \subset \mathrm{INVAR}$	collection of model invariants
$evt \subset \mathrm{EVENT}$	event collection

The general form of an Event-B event is

$$nom\ \mathbf{ref}\ par \quad = \quad \begin{array}{l} \textbf{ANY}\ arg\ \textbf{WHERE} \\ \quad grd \\ \textbf{THEN} \\ \quad act \\ \textbf{END} \end{array}$$

An Event-B event declaration comprises a list of parameters, a guard and an action. It is convenient to represent an event guard as a conjunction of predicates and an event action as a parallel composition of actions. An event may also carry the **refines** attribute which lists the set of abstract events refined by the event. We represent an event as an object with the following attributes:

$nom \in \text{LABEL}$	event name
$par \in \mathcal{P}(\text{LABEL})$	names of refined events
$arg \subset \text{PARAM}$	event parameters
$grd \subset \text{GUARD}$	event guards
$act \subset \text{ACTION}$	event actions

An Event-B model description, introduced above, references only event identifiers, not event descriptions. Description of an event is provided by the $mevt$ function:

$$mevt : \text{EVENT} \rightarrowtail \text{LABEL} \times \mathcal{P}(\text{PARAM}) \times \mathcal{P}(\text{GUARD}) \times \mathcal{P}(\text{ACTION})$$

where $evt \overset{\text{df}}{=} dom(mevt)$.

Event-B action is a tuple of variables, a substitution style and an expression computing new variable states:

var	assigned variables
sty	assigment style
exp	assigment expression

Action descriptions are provided by the following function:

$$mact : act \rightarrowtail \mathcal{P}(\text{VAR}) \times \text{STYLE} \times \text{EXPR}$$

The act attribute of an event must be contained in the domain of the $mact$ function. It is possible, however, that some actions described by the function are not used anywhere in a model.

An Event-B variable declaration is split into three model parts. The **variables** section declares the names of variables. The typing predicates for model variables are provided by the **invariant** section. Finally, all system variables must be initialised by the *initialisation* event:

...
VARIABLES
    ..., *lbl* (identifier), ...
INVARIANT
    ... $\wedge$ *typ* (typing predicate) $\wedge$ ...
INITIALISATION
    ...$\|act$ (initialisation action)$\|$...
...

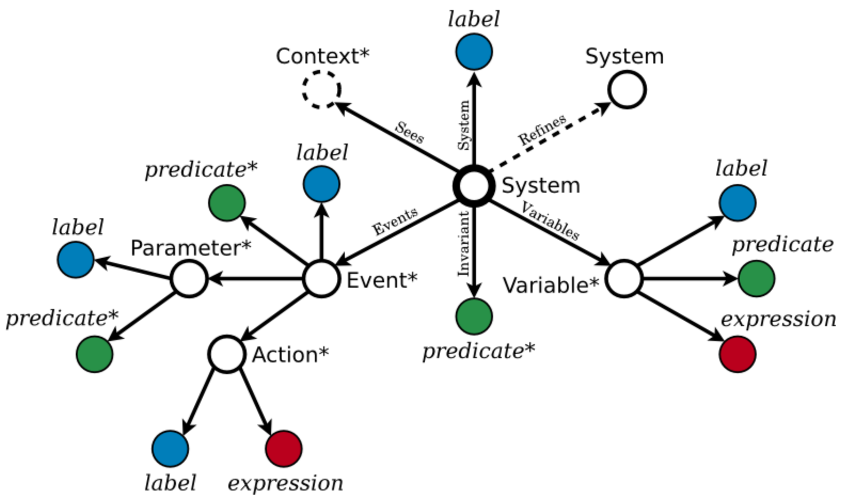


Figure 3.2: Event-B Model syntactic elements.

We find it more convenient to combine different parts of a variable description into a single object with a variable name, a typing invariant and an initialisation action. Because of this, there is no need for an initialisation event although its equivalent can be computed from variable descriptions. A variable is described by a label (the variable name), an initialisation action and a typing predicate:

nom	$nom \in \text{LABEL}$	identifier
act	$act \in \text{ACTION}$	initialisation action
typ	$type \in \text{TYPE}$	typing predicate

Initialisation action of a variable must update the variable it initialises:

$$v = (n, a, t) \wedge a = (av, as, ae) \Rightarrow v \in av$$

Several variables may share an initialisation action. This happens when an action is a non-deterministic substitution mentioning several variable. For instance, action

$$a, b : | a > b$$

is a shared initialisation action for variables a and b. Variable descriptions are stored in the $mvar$ function:

$$mvar : \text{VAR} \nrightarrow \text{LABEL} \times \text{ACTION} \times \text{TYPE}$$

and $var \overset{\text{df}}{=} dom(mvar)$.

An event parameter is characterised by a name and a typing predicate;

nom	label (identifier)
typ	typing predicate

Descriptions of event parameters are provided by function $marg$:

$$marg : arg \nrightarrow \text{LABEL} \times \text{TYPE}$$

An attribute arg of an event tuple must be in the domain of $marg$.

3.5.1 Useful Definitions

To simplify the formulation of the Event-B model transformations, we introduce a number of notational shortcuts. An abstract model is referred to as $m_0 = (var_0, inv_0, evt_0)$ (and $mvar_0$, $mevt_0$ for variable and event descriptions) while the current model is described by $m = (var, inv, evt)$.

An abstract model is an input of a model transformation, that is the model transformed by the model transformation. Current model is the result of an application of a model transformation to some abstract model. Hence, for the two models m_0 and m, there is some model transformation md, such that $m = md(m_0)$.

Individual elements of a tuple, characterising a model element, can be referred to by their names using the following notation:

$$e.\text{prop}$$

For example, a name of event e is $e.\text{nom}$. The same, but for the elements of an abstract model, is written as

$$\hat{e}.\text{prop}$$

Tuple (L_v, A_v, T_v) describes some arbitrary variable and tuple (L_p, T_p) is some arbitrary parameter. Label L_e is used as a default event name.

Fresh labels for events and variables are computed by removing the two predefined constant labels, names of all the events, variables and arguments from the labels carrier set:

$$\begin{aligned} \textit{FreshNames} \overset{\text{df}}{=} \; & \text{LABEL} \setminus (\{L_v, L_e\} \cup (\textstyle\bigcup_{v \in var}\{v.\text{nom}\}) \cup \\ & \textstyle\bigcup_{e \in evt}(e.\text{nom} \cup (\bigcup_{a \in e.\text{arg}}\{a.\text{nom}\}))) \end{aligned}$$

An argument name has only to be unique within the scope of a given event:

$$\begin{array}{rcl} \mathit{FreshArgNames}(e) & \overset{\text{df}}{=} & \text{LABEL} \setminus (\{nom, par\} \cup (\bigcup_{v \in var}\{v.\text{nom}\}) \cup \\ & & \{e.\text{nom}\} \cup (\bigcup_{a \in e.\text{arg}}\{a.\text{nom}\})) \end{array}$$

The following shortcuts are used to find whether a variable or an event is inherited from an abstract model:

$$\begin{array}{rcl} isAbstrVar(v) & \overset{\text{df}}{=} & v \in dom(mvar_0) \\ isAbstrEvt(e) & \overset{\text{df}}{=} & e \in dom(mevt_0) \end{array}$$

The non-trivial Event-B model transformation requirements are the conditions derived from the Event-B well-formedness and refinement proof obligations. For more compact model transformation definitions we declare such conditions as a list of parametrised predicates.

For each declaration of a variable it is required to demonstrate that the variable initialisation action satisfies the feasibility and invariant preservation properties of Event-B:

$$\begin{array}{ll} \mathsf{PAT\_FIS\_INI}(s) & \exists w' \cdot s(w') \\ \mathsf{PAT\_INV\_INI}(s, j) & s(w') \implies j(v, w') \\ \mathsf{PAT\_RIN\_INV}(s, r, j) & I(v) \wedge j(v, w) \wedge s(w') \implies \exists v' \cdot (r(v') \wedge j(v, w')) \end{array}$$

where $s(w')$ is a before-after initialisation predicate providing a value for concrete model variables w, $j(v, w)$ (supplied as a parameter here) is a concrete model invariant which includes a gluing invariant linking abstract variables v and concrete variables w. $I(v)$ is an abstract invariant and $RI(v)$ is an abstract before-after initialisation predicate. Note, that we do rely on the fact that the abstract input specification is well-formed (in the sense of Event-B model well-formedness). The consistency properties of an abstract model can be used as hypothesis to discharge the current model consistency and refinement proof obligations. A change in an event inherited from an abstract model gives rise to the following conditions:

$$\begin{array}{ll} \mathsf{PAT\_REF\_FIS}(s) & I(v) \wedge J(v, w) \wedge H(w) \Rightarrow \exists w' \cdot s(w, w') \\ \mathsf{PAT\_REF\_GRD}(g, h) & I(v) \wedge J(v, w) \wedge h(w) \Rightarrow g(v) \\ \mathsf{PAT\_REF\_INV}(s, a) & I(v) \wedge J(v, w) \wedge H(w) \wedge s(w, w') \Rightarrow \\ & \exists v' \cdot (a(v, v') \wedge J(v', w')) \\ & \text{provided } I(v) \wedge G(v) \wedge R(v, v') \implies I(v') \end{array}$$

where g, G and h, H are the abstract and concrete guards and $s(w, w')$ and $R(v, v')$ are the abstract and concrete before-after predicates. The first condition is a form of the Event-B proof obligation requiring an action to be able to provide some result under the given invariant and guard. The guard refinement PAT_REF_GRD states that a guard of a refined event strengthens the guard of the corresponding abstract event. Finally, the invariant preservation condition requires an action to satisfy a model invariant while refining the abstract action.

Actions of new events must preserve the concrete model invariant:

$$\mathsf{PAT\_NEW\_INV}(s) \quad I(v) \land J(v,w) \land H(w) \land s(w,w') \Rightarrow J(v,w')$$

The non-divergence must be demonstrated for all the new events. For this, one has to demonstrate that there exists an expression V, called variant, decrement by all the new events:

$$\begin{array}{ll}\mathsf{PAT\_NEW\_DIV} & I(v) \land J(v,w) \land H(w) \land S(w,w') \Rightarrow \\ & V(w) \in \mathbb{N} \land V(w') < V(w)\end{array}$$

where S is a concrete before-after predicate.

A concrete model invariant is the conjunction of invariant predicates and variable typing predicates:

$$J(v,w) \quad \overset{\text{df}}{=} \quad (\textstyle\bigwedge_{i \in inv} i) \land (\bigwedge_{v.\text{nom} \in dom(mvar)} v.\text{typ})$$

Guard of an event e is the conjunction of individual events guards and typing predicates of event parameters:

$$H(w) \quad \overset{\text{df}}{=} \quad (\textstyle\bigwedge_{g \in e.\text{grd}} g) \land (\bigwedge_{a \in e.\text{arg}} (a.\text{nom} \in a.\text{typ}))$$

Sometimes we need to construct a new invariant where a variable typing predicate replaced with some new predicate. For variable rv and a new typing predicate rt this is expressed as

$$J\_vt(rv, rt) \quad \overset{\text{df}}{=} \quad (\textstyle\bigwedge_{i \in inv} i) \land (\bigwedge_{v \in dom(mvar) \land v \neq rv} (v.\text{nom} \in v.\text{typ})) \land (rv \in rt)$$

and a similar condition, but for an event guard, is

$$G\_vt(rv, rt) \quad \overset{\text{df}}{=} \quad (\textstyle\bigwedge_{g \in e.\text{grd}} g) \land (\bigwedge_{a \in e.\text{arg} \land a \neq rv} (a.\text{nom} \in a.\text{typ})) \land rv \in rt$$

3.5.2 Event-B Model Facets

In the context of Event-B, some predicates describing model input and output classes can be presented in a more legible notation, resembling the syntax of Event-B models. We call such descriptions *facets*. A facet uses Event-B syntax to describe a model by listing the required model elements and their properties.

Any facet can be mapped into a predicate describing a model class. Some predicates, however, cannot be represented using facets. For example, facets cannot be used to express removal of a syntactic element (e.g., removal of an operation from a model). The following examples demonstrate the syntax of Event-B facets and their mappings into the predicate form (here $\equiv$ denotes equivalence):

- a model must declare at least one variable

$$v \in var \quad \equiv \quad \begin{pmatrix} \textbf{FACET } t_0 \\ \textbf{VARIABLES } ?v \end{pmatrix}$$

- the above, and the variable must be a natural

$$v \in var \wedge v.\text{typ} = NAT \quad \equiv \quad \begin{pmatrix} \textbf{FACET } t_1 \\ \textbf{VARIABLES } ?v \\ \textbf{INVARIANT } ?v \in \mathbb{N} \end{pmatrix}$$

- the above, and there must be an event which increments the variable

$$\begin{pmatrix} v \in var \wedge v.\text{typ} = NAT \\ e \in evt \wedge a \in e.\text{act} \\ a = (\{v\}, (:=), (v+1)) \end{pmatrix} \quad \equiv \quad \begin{pmatrix} \textbf{FACET } t_2 \\ \textbf{VARIABLES } ?v \\ \textbf{INVARIANT } ?v \in \mathbb{N} \\ \textbf{INITIALISATION } ?v :\in \mathbb{N} \\ \textbf{EVENTS} \\ \quad ?e \;=\; \textbf{BEGIN} \\ \qquad\qquad ?v := ?v + 1 \\ \qquad\qquad \textbf{END} \end{pmatrix}$$

Facets are conjunctions of positive statements, a facet cannot express a requirement that something is not present.

3.5.3 Transformation Definitions

In this section we overview Event-B model transformations. The list does not include derivative transformations that can be obtained by reversing a direct transformations. The list with the definitions for Event-B model transformations can be found in Appendix A.

- $\mathsf{addvar}(v)$ - adds a new system variable. The new variable has some arbitrary label, type and initialisation. Addition of a new variable always results in a valid model refinement; reversible.

$$\mathsf{addvar}(v) \quad \equiv \quad \begin{array}{l} \textsf{requirements} \\ \quad v \in (\text{VAR} \setminus dom(mvar)) \\ \textsf{effect} \\ \quad mvar' = mvar \cup \{v \mapsto (L_v, A_v, T_v)\} \\ \textsf{scope} \\ \quad v, L_v \\ \textsf{ref} \\ \quad true \end{array}$$

The inverse of this transformation removes an existing variable from a model

$$\mathsf{addvar}^{-1}(v) \equiv \begin{array}{l} \textsf{requirements} \\ \quad v \in dom(mvar) \\ \textsf{effect} \\ \quad mvar' = \{v\} \rhd mvar \\ \textsf{scope} \\ \quad \star v \\ \textsf{ref} \\ \quad \mathit{false} \end{array}$$

- varlabel(v, nl) - changes name of a system variable; reversible.
- newvaraction(v, na) - provides a new initialisation action for a system variable.
- newvartype(v, nt) - changes the typing predicate of a variable.
- addpar(a) - adds a new event parameter; reversible.
- parlabel(a, nl) - changes event parameter name.
- newpartype(a, nt) - changes event parameter type.
- addevent(e, l) - adds a new event with no parameters, empty guard and no actions; reversible.
- refines(e, nr) - adds an abstract event to the *refines* list of a concrete event; reversible.
- newevtlabel(e, nl) - renames an event.
- addguard(e, h) - adds a new guard predicate; reversible.
- addparam(e, par) - adds an event parameter; reversible.
- addaction(e, act) - adds a new action; reversible.

$$\mathsf{addaction}(e, act) \equiv \begin{array}{l} \textsf{requirements} \\ \quad e \in dom(mevt) \land e = (l, p, ga) \land act \in dom(mact) \land \\ \quad \mathsf{PAT\_REF\_FIS}([\{act\}]) \land \\ \quad isAbstrEvt(e) \Rightarrow \mathsf{PAT\_REF\_INV}([a \cup \{act\}], [\hat{e}.\mathrm{act}]) \land \\ \quad \neg isAbstrEvt(e) \Rightarrow \mathsf{PAT\_NEW\_INV}([act]) \\ \textsf{effect} \\ \quad mevt' = mevt \lhd\!\!\!- \{e \mapsto (l, p, g, a \cup \{act\})\} \\ \textsf{scope} \\ \quad e, act.\mathrm{var} \\ \textsf{ref} \\ \quad \mathit{true} \end{array}$$

- $\mathsf{addinv}(i)$ - adds a new model invariant; reversible.
- $\mathsf{delinv}(i)$ - removes part of a model invariant.
- $\mathsf{newaction}(a, v, s, e)$ - constructs a new action with the given variable, style and expression; reversible.
- $\mathsf{newactionvar}(a, nv)$ - adds a new variable to an action; reversible.
- $\mathsf{newactionsty}(a, ns)$ - changes substitution style of an action.
- $\mathsf{newactionexp}(a, ne)$ - changes action expression.

3.5.4 Example

In this example we use a composition of model transformations to construct a simple refinement step. The refinement introduces a new integer variable and adds an action to an existing event to increment the new variable. The event is constrained with a predicate limiting the number of increments to 10.

The complex model transformation achieving this is defined as follows

$$addinc \overset{\mathrm{df}}{=} \begin{pmatrix} \mathsf{addvar}(v); \mathsf{varlabel}(v, u); \\ \qquad \mathsf{newaction}(a, v, (:\in), (0 \ldots 5)); \\ \qquad\qquad \mathsf{newvaraction}(v, a); \\ \qquad\qquad \mathsf{newvartype}(v, \mathbb{Z}); \\ \qquad\qquad\qquad \mathsf{newaction}(a_1, v, (:=), (u + 1)); \\ \qquad\qquad\qquad\qquad \mathsf{addaction}(e, a_1); \\ \qquad\qquad\qquad \mathsf{addguard}(e, (u < 10)) \end{pmatrix}$$

Note the use of variable label and variable object. v is a variable object and name v is only known within the scope of the transformation composition. Label u is the name by which the variable v is known in the refined model. Thus, the increment of v is done by assigning expression $u + 1$. Further we may ignore this distinction, especially in cases when there is a completely formal, tool-based description.

Below is an example of an acceptable input specification along with a result of the application of the model transformation to this model:

```
SYSTEM $m0$
VARIABLES
    $s$
INVARIANT
    $s \in \mathbb{N}$
INITIALISATION
    $s := 0$
EVENTS
    $a$  =  BEGIN
              $s := s + 1$
            END
    $b$  =  BEGIN
              $s := s + 2$
            END
```

```
SYSTEM $m0\_addinc$
REFINES $m0$
VARIABLES
    $s, q$
INVARIANT
    $s \in \mathbb{N}$
    $q \in \mathbb{Z}$
INITIALISATION
    $s := 0$
    $q :\in 0..5$
EVENTS
    $a$  =  WHEN
              $q < 10$
            THEN
              $s := s + 1$
              $q := q + 1$
            END
    $b$  =  BEGIN
              $s := s + 2$
            END
```

Note that the variable name q is not defined by the transformation: it is some arbitrary fresh name provided as a value for label u.

3.6 Summary

In this chapter we have discussed the concept of model transformation, showed how to describe transformations and defined a set of model transformations for the Event-B method. We have discussed how to construct inverse transformations and how to compose transformations.

Model transformations is an active research area in the context of model driven software development [76] where a large number of model types (mainly informal) are used to specify different system aspects, from high-level organisational properties of a business process up to the intricacies of a physical level communication protocol. To keep different models in agreement it is important to be able to do conversion from one model type into another. The importance of well-established and generally accepted model transformation steps is also widely recognised [77, 78].

A complex refinement step is likely to require a large number of model transformations. In the next section we discuss a more general mechanism of patterns - compositions of model transformations that always produce correct refinement or abstraction steps. While model transformations are concerned with basic transformation rules, refinement patterns are suitable for description of reoccurring design decisions. In addition, unlike a model transformation, that, in a general case, only changes a model, a refinement pattern always constructs a complete model refinement. By separating definition of model transformations from formulation of refinement patterns we are able to construct a method-neutral patterns framework.

Chapter 4

Refinement Patterns

4.1 Introduction

Model transformations, discussed in the previous chapter, are too low-level to be of a major assistance to a modeller. In this chapter, we introduce a language to combine model transformations into *patterns*. A pattern describes a reusable and practically important case of refinement or abstraction. For many patterns, we are able to statically demonstrate correctness in the sense that the produced model is always a valid abstraction or a refinement. There are two major pattern kinds. Patterns converting abstract models into concrete models are called *refinement patterns* [79–81]. An *abstraction pattern* computes an abstract model from its concrete counterpart.

One of the obstacles to the wide adoption of formal methods is that while there are plenty of programmers capable of writing complex software using programming languages, there are few modelling experts capable of handling the mathematical notation used in formal models. Patterns give a psychological advantage - low-level details and mathematics are hidden by an easy to comprehend high-level pattern description. A pattern instantiation requires no proofs at all: to guarantee the correctness of a constructed refinement or an abstraction, it is enough to demonstrate the pattern applicability conditions, required to instantiate a pattern.

With a high-level programming language and a collection of code libraries, programmers can relatively quickly construct large programs through the reuse of a vast amount of high-quality third-party code. Formal methods and supporting tools are focused on the act of verification and thus do not offer much support for model construction. A pattern can be seen as an analogue of a reusable procedure imported from a third-party code library.

This chapter describes the pattern language, discusses the benefits of applying such patterns in formal developments, formally describes the patterns instantiation mechanism and the proof theory used to establish pattern correctness. In the development of the pattern language we tried to struck a balance between expressiveness and verifiability. We would like to able to proof pattern correctness once and for all possible pattern instantiations but we also require that a pattern instantiation is handled automatically by a tool. The chapter includes the continuation of the tree model example which is extended with the definitions of refinement patterns.

4.2 Motivation

Modelling is often seen as an accessory to programming, where a model is used to construct a system and then is discarded. Although modelling is one of the more expensive stages of a system construction (provided it relies on formal modelling), models are rarely reused and adapting a legacy formal development to a new system design is nearly impossible. We propose taking a different view on the modelling process. We see a formal development not only as a way to arrive to a final model but also as a process creating important, reusable artifacts. In our approach, the model creation process is understood as a chain of design decisions. Many such decisions are specific to a problem being solved, but there are often some general ones that could be reused within the same or a different development. In methods supporting refinement, such as the Refinement Calculus [35], Z [13] and the B Method [16], a formal development is a chain (or, in a more general case, a tree) of models, linked by the refinement relation. The development process focuses on the construction of individual models. Once a model is ready, a modeller must prove the refinement relation in respect to the model abstraction. The whole process is repeated as long as needed, e.g. until an executable code can be generated (Figure 4.1).

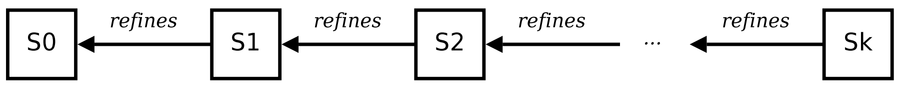


Figure 4.1: In a chain of refinements, each successive model is a detalisation of its abstraction.

In our approach the focus is shifted from the construction of a model refinement to the formulation of steps that lead to a refined model. We introduce a new entity in the formal development process which sole purpose is to describe refinement steps leading from an abstract model to a refined model. We call such entity a *refinement pattern* (first introduced in [79]) and understand it as a function which, when applied to an abstract model, yields a concrete model. The name *pattern* emphasizes the fact that such function is a reusable design solution applicable in a range of contexts, much like design patterns developed for object-oriented programming [39]. We also study the correlative of a refinement pattern - an abstraction pattern - that can be used to reverse engineer a formal development, or possibly, help to construct refinement patterns in cases where an abstraction pattern is easier to specify. We believe that the patterns mechanism offers a number of advantages:

- shift of perspective - normally a modeller sees and works with a whole model corresponding to the current refinement step. In our approach we let a modeller to focus on the description of a particular refinement transformation and, thus, focus on a specific part of a system design;
- better scalability - a realistic system model is inevitably large. Our approach addresses the scalability problem by representing detailed and complex development as a succession of loosely coupled or completely independent patterns;
- reusability - patterns can be used in more than one context within the same development and can be applied in completely unrelated developments;
- better structuring - a pattern deals with a specific aspect of a system functionality. In many cases, patterns can be developed independently and simultaneously and this does not require model decomposition;
- less proofs - proofs done for a refinement pattern are automatically reused each time a pattern is included into a development. In a large development, this results in a considerable proofs economy.

In addition, being completely self-sufficient, the pattern mechanism facilitates the exchange of ideas between designers. A pattern can be transferred, purchased or sold. A development can be, at least partially, constructed from ready-made third-party refinement or abstraction patterns, which, ultimately, means reduced modelling costs and a wider adoption of formal modelling.

4.3 A Definition of Refinement Patterns

The fact that specification s is refined by s' is denoted as $s \sqsubseteq s'$. The $\sqsubseteq$ relation is reflexive, transitive and antisymmetric. To simplify the discussion, we often omit the mentioning of abstraction patterns when their properties mirror the corresponding properties of refinement patterns.

Definition A refinement pattern is a function producing a refinement for some input model and a pattern configuration:

$rpatt : S \times C \rightarrow\!\!\!\!\!\!+ \ S$

where S is a universe of models and C are pattern configurations.

The addition of a configuration helps to abstract away from the specifics of a particular input model and reuse patterns in different developments. Pattern configuration also allows a modeller to adjust pattern to specific needs when a pattern is added to a development. The ability to provide a pattern configuration also helps to resolve a non-deterministic choice during a pattern application.

A refinement pattern is expected to produce a valid model refinement. Formally this is stated as pattern correctness requirement. Correctness of a pattern is demonstrated by proving that it always produces valid refinement or abstraction models.

Definition A refinement pattern is correct if it produces a valid refinement for any accepted combination of an input model and pattern configuration:

$\forall s, c \cdot ((s, c) \in dom(rpatt) \Rightarrow s \sqsubseteq rpatt(s, c))$

In its current form, the correctness definition does not lead to any practical means of pattern verification. Later in this chapter we, having defined a simple pattern language, we able to generate a list of more tractable conditions.

An example of a trivial refinement pattern is pattern mapping a model into itself: $\mathbf{id}_S$. This pattern also doubles as an abstraction pattern and it is an example of a pattern that does not use configuration.

4.4 Language of Refinement Patterns

Pattern is a named entity. A pattern declaration starts with a construct declaring the pattern name:

pattern p

where p is the name of the declared pattern.

The body of a pattern is a combination of model transformations glued together using a number of control structures. The simplest form of a pattern is a single model transformation rule:

$$mdr(p)$$

where p is a pattern configuration and an input model for mdr is assumed as an implicit parameter.

The two main operations used in a pattern declaration are the sequential and parallel compositions. In a pattern specification we use indentations to denote blocks of rules and the style of composition operator used to connect the rules. The sequential composition requires the next rule to be written below and indented to the right. Rule

$$\begin{array}{ll} r_1 & \\ & r_2 \end{array}$$

prescribes application of rule r_2 after r_1. If r_1 cannot be applied then r_2 is not used as well. The sequential composition is normally used to put together rules that depend on each other. More precisely, the second of two rules, is applied to the model constructed by applied the first rule. Elements with the same indentation level are said to be parallel. The parallel composition is denoted by equal indentation of the composed rules:

$$\begin{array}{l} r_1 \\ r_2 \end{array}$$

The rule above has two independent rules, r_1 and r_2, which are supposed to work on disjoint model parts and, thus, can be applied in any order. It is the requirement that the parallel composition Is only used to connect rules operating on strictly disjoint model parts. A pattern rule constructed using the sequential or the parallel compositions can be nested in another rule:

$$\begin{array}{ll} r_1 & \\ & r_a \\ & r_b \end{array}$$

Here the parallel composition of two rules is sequentially composed with another rule. In this pattern, rules r_a and r_b can be applied in any order but only after rule r_1. The sequential composition binds stronger than the parallel composition and brackets are used when an instance of the parallel composition must take precedence

$$\begin{array}{ll} \begin{pmatrix} r_a \\ r_b \end{pmatrix} & \\ & r_1 \end{array} \neq \begin{array}{ll} r_a & \\ r_b & \\ & r_1 \end{array} = \begin{array}{l} r_a \\ \begin{pmatrix} r_b & \\ & r_1 \end{pmatrix} \end{array}$$

The role of the parallel composition is the simplification of the pattern correctness analysis. Later we show that to prove that a given instance of the parallel composition is correct it is enough to independently show the correctness of each constituent rule.

With the parallel composition, one has to make sure that the composed rules transform non-overlapping model parts and the order of the rule applications does not affect the overall pattern result. It is possible to deduce this property directly from model transformation rules but we prefer to construct the list of model elements a model transformation updates or depends on when the transformation is formulated. The results in the following extended syntax for model transformations:

```
name($p$)  ≡  requirements
                $c(s, p)$
              effect
                $s' = r(s, p)$
              scope
                $e$
              ref
                $r$
```

To be able to construct practically important patterns we have to provide a mechanism for matching and selecting model parts that need to be transformed. This is achieved with the following construct:

```
forall $a$ with $p$ where
    $P(a, p)$
do
    $r(a, p)$
end
```

where a is a vector of local variables bound by the constraining predicate $P(a, p)$ (the predicate also depends on an implicit input model parameter), p is a vector of local parameters and r is a nested rule. The rule r is applied to every possible value (a, p), as defined by the predicate P. Local variables a are selected automatically during a pattern instantiation while the values for p are provided by a modeller, once for each *forall* construct. In other words, a contained rule r is applied to all possible values of a but each times uses the same vector p. The collection of all such parameters is the configuration of a pattern.

The *forall* construct essentially describes a generalised version of parallel composition. Thus, where we would have to write a long list of similar transformations -

$$\begin{array}{l} md(1) \\ md(2) \\ \ldots \\ md(32) \\ md(33) \end{array}$$

- we can use a single *forall* statement:

```
forall $i$ where
    $i \in 1..33$
do
    $md(i)$
end
```

There are several shortcut forms accounting for the cases when some of the statement parts are omitted. The first such shortcut has no local variables but uses parameters:

```
with $p$ where
    $P(p)$
do
    $r(p)$
end
```

With no parameters and free variables, *forall* degrades into a simple *when* construct:

```
when
    $P$
do
    $r$
end
```

Being one of the possible forms of a pattern rule, *forall* can be nested in another *forall* construct. It is often convinient to use the *with* shortcut form to declare parameters shared by several rules:

```
with $p$ where
    $P(p)$
do
    $r_1(p)$
    $r_2(p)$
    ...
end
```

To make patterns more readable, this case has its own shortcut form. This construct is normally placed immediately after the pattern declaration clause:

```
pattern $p$
  req_$name$ $P$
```

Here P is a predicate. The free variables of this predicate are the declared parameters and the *_name* part is a short description of the declaration. The equivalent *with* declaration is

```
with $freevar(P)$ where
    $P$
do
    ...
end
```

where $freevar(P)$ are the free variables of P.

The nesting of rules controls the visibility of names. Children of a rule can see all the parameters and local variables available to their parent. This relation is transitive - a rule sees all the parameters available to its parent rule. Parameters declared with req_ are seen by all the rules of a pattern.

The pattern language we have discussed is method-neutral. We believe that the framework can be used to describe transformation patterns for a wide range of specification methods. To relate the framework to a specific formalism (or a family of formalisms) we have to import a library of model transformations. Symmerically, to bring the support for the pattern mechanism into a formal method it is enough to construct a library of model transformations that can be used to describe interesting refinement (or abstraction) cases.

If a model transformation library is specific to a given formalism, the corresponding patterns are applicable only to that particular formalism. A library, however, can be made generic so that the patterns based on this library are applicable to models of several formalisms. Such library would use abstract model transformation definitions and provide mappings into actual model transformations of concrete formalisms. This could help decouple the discussion of a modelling formalism transformations from specification of high-level patterns. As an example, we can declare an abstract transformation rule adding an event to a model and provide mappings into concrete realisations of this transformation for a range of formalisms that use the event concept.

4.4.1 Applying Patterns

A pattern is applied to a model to produce a new model. A refinement pattern takes an abstract model and produces its concrete version. An abstraction pattern works in the other direction. Interpretation of a pattern is provided by a function of the following form:

$$\mathbf{eff} : \mathrm{MODEL} \times \mathrm{RULE} \rightarrow \mathrm{MODEL}$$

In this definition we choose to omit the pattern configuration which provides pattern instantiation parameters. We assume that we have a complete configuration before applying a pattern and, for legibility reasons, assume pattern configuration as an implicit parameter of **eff**.

The effect of the sequential composition of two rules is computed by applying the the second rule to the result of the first rule:

$$\mathbf{eff}\left(s, \begin{matrix} a & \\ & b \end{matrix}\right) = \mathbf{eff}(\mathbf{eff}(s, a), b)$$

For the parallel composition the effect is computed in the same manner but the order of the rule applications should not affect the overall result:

$$\mathbf{eff}\left(s, \begin{matrix} a \\ b \end{matrix}\right) = \mathbf{eff}(\mathbf{eff}(s, a), b) = \mathbf{eff}(\mathbf{eff}(s, b), a)$$

The *forall* construct applies the body rule to all the possible values of local variables:

$$\mathbf{eff}\left(s, \begin{array}{l} \mathsf{forall}\ a\ \mathbf{with}\ p\ \mathsf{where} \\ \quad P(a,p) \\ \mathsf{do} \\ \quad r(a,p) \\ \mathsf{end} \end{array}\right) = \mathbf{eff}\left(s, \begin{array}{l} r(q,p) \\ \quad \mathsf{forall}\ a\ \mathbf{with}\ p\ \mathsf{where} \\ \quad\quad P(a,p) \land a \neq q \\ \quad \mathsf{do} \\ \quad\quad r(a,p) \\ \quad \mathsf{end} \end{array}\right)$$

where $P(a, p)$ holds for $a = q$. Note that at each iteration the constraining predicate is further restricted by a conjunction eliminating one of the "old" values. This ensures termination and complete coverage of the parameter space for $P(a, p)$ characterising finite sets.

The effect of a model transformation is computed by applying the model transformation to an input model:

$$\textbf{eff}\,(s, mdr(p)) = mdr(p)$$

where transformation rule $mdr(p)$ also takes input model s as an implicit parameter.

4.4.2 Pattern Correctness

A pattern correctness can be demonstrated in a number of ways [82]. The simplest approach is to rely on the normal means of proving the refinement (abstraction) relation between two models and to not try to analyse correctness of a pattern itself. In other words, each time a pattern is applied, the result must be shown to be a valid refinement using the refinement relation conditions of the applied formalism.

We are not going to demonstrate formally that the given set of rules is enough to establish pattern correctness. We believe such a rigorous justification is outside of the scope of this work and is one directions for the future work on the patterns mechanism.

The second approach is to statically demonstrate that a pattern result is correct in the sense of the pattern correctness definition above. Sometimes this is not possible. For this we introduce the mechanism of assertions and rely on a combination of static pattern analysis and theorems generated per each pattern instantiation.

A pattern is correct if, for all the constituent rules, the following conditions are satisfied.

Computability A patterns is a procedure that should be processable by a software tool. Thus, we need to make sure that a pattern is executable. A problem can arise when a constraining predicate of the *forall* construct describes an infinitely large set of parameters. Such pattern, although possibly useful, makes no sense to a tool. We express this by stating that a constraining predicate is a characteristic function of a finite set:

$$finite(\{(a) \mid P(a)\})$$

Conflict-freeness The effect of application of a parallel composition of rules should not dependent on the order of rule applications. This property holds when the rules are applied to disjoint model parts. We formulate this condition by stating that the scopes of all the nested model transformations are non-overlapping. For any parallel rule composition:

$$\begin{array}{l} r_1 \\ r_2 \\ \dots \\ r_k \end{array}$$

the following condition must be satisfied

$$\forall i, j \cdot (i \in 1..k \wedge j \in 1..k \wedge i \neq j \Rightarrow \mathit{inter}(\mathit{scope}(r_i), \mathit{scope}(r_j)) = \oslash)$$

where the inter predicate computes the intersection of two scopes (see the definition in Appendix A) and the scope function computes the scope of a rule by aggregating scopes of its constituent rules, as follows

$$\begin{array}{lcr}
\mathit{scope}\begin{pmatrix} a & \\ & b \end{pmatrix} & = & \mathit{scope}(a) \cup \mathit{scope}(b) \\
\mathit{scope}\begin{pmatrix} a \\ b \end{pmatrix} & = & \mathit{scope}(a) \cup \mathit{scope}(b) \\
\mathit{scope}\,(mdr(p)) & = & \mathit{scope}(mdr) \\
\mathit{scope}\begin{pmatrix} \text{forall } a \ \mathbf{with}\ p \text{ where} \\ \quad P(a,p) \\ \text{do} \\ \quad r(a,p) \\ \text{end} \end{pmatrix} & = & \mathit{scope}(r)
\end{array}$$

where $\mathit{scope}(mdr)$ returns the scope of a model transformation. The union of scopes is defined as

$$\mathit{scope}((r_1, w_1)) \cup \mathit{scope}((r_2, w_2)) = (r_1 \cup r_2, w_1 \cup w_2)$$

Well-formedness A pattern rule may call its child rule only when the child rule precondition is satisfied. For each model transformation used in a pattern body it is required to demonstrate that the transformation requirement is satisfied for all the contexts in which the model transformation may be invoked. We state this by requiring that the context of a parent rule implies the requirement of a model transformation under analysis. Thus, for a rule

```
$q$
        forall $a$ with $p$ where
            $P(a, p)$
        do
            $mdr(a, p)$
        end
```

we have to demonstrate that

$$\forall s, a \cdot (\mathbf{incl}(q)(s) \wedge \mathbf{eff}(s, q) \wedge P(a, p) \Rightarrow \mathbf{req}(mdr)(a, p))$$

Here $in(q)(s)$ states that model s is in the input model class of the parent rule q; $\mathbf{eff}(s, q)$ computes the result of the application of q to s; $req(mdr)(a, p)$ is the input model class of the mdr model transformation with the parameters instantiated to values of p and a. Note that s is an implicit parameter of mdr.

We know that for any s the effect of pattern rule computation satisfies the output model class predicate

$$\mathbf{outcl}(q)(\mathbf{eff}(s, q)) \equiv true$$

The condition above can be replaced with a stronger condition requiring that output model class of the parent rule, combined with the constraining predicate, is stronger than the model transformation requirement

$$\mathbf{outcl}(q) \wedge P(a, p) \Rightarrow \mathbf{req}(mdr)(a, p)$$

The former version is more suitable for generating proof obligations during a pattern instantiation. The latter imposes a stronger condition but has to be demonstrated only once for a pattern instance.

For the case when there is no *forall* statement for the analysed model transformation, the condition above is simplified to

$$\forall s \cdot (\mathbf{incl}(q)(s) \wedge \mathbf{eff}(s, q) \Rightarrow \mathbf{req}(mdr)(c))$$

or

$$\mathbf{outcl}(q) \Rightarrow \mathbf{req}(mdr)(c))$$

where c are some constant parameters. In absence of any parent rule, the constraining predicate alone must ensure that a model transformation requirement is satisfied

$$P(a,p) \Rightarrow \mathbf{req}(mdr)(a,p)$$

Refinement or Abstraction For a model transformation that does not construct a valid model refinement or abstraction, we have to demonstrate that a combination of the transformation with other patterns rules results in a valid model refinement rule. This can be done by finding a containing subset of pattern rules that, together with the model transformation being examined, results in a valid refinement or abstraction step.

To prove that the refinement condition is satisfied in respect to some model transformation mdr, we analyse the rule together with its neighbour rules -

```
pattern $patt$
  $q$
        $mdr(p)$
              $r$
```

- and try to demonstrate that the combined effect of these rules yields a valid model refinement pattern:

$$\forall s \cdot (in(q)(s) \Rightarrow s \sqsubseteq \mathbf{eff}(\mathbf{eff}(\mathbf{eff}(s,q), mdr(p)), r))$$

where s is an abstract model transformed by the pattern; q and r are the pattern rule sequentially composed with the analysed model transformation. An expanded form of the condition above, more suitable for doing once-for-all proofs , is

$$\begin{pmatrix} \mathbf{incl}(q)(s) \wedge s' = \mathbf{eff}(q)(s) \\ \mathbf{req}(mdr)(s') \wedge s'' = \mathbf{eff}(s', mdr(p)) \\ \mathbf{incl}(r)(s'') \wedge s''' = \mathbf{eff}(r)(s'') \end{pmatrix} \Rightarrow s \sqsubseteq s'''$$

From our experience with several large-scale refinement patterns (see Chpater 6), an instantiation of these conditions results in many redundant hypothesis on the left-hand side and usually the rule can be reduced to a compact and manageable form. In few cases, when this was not true and we had to analyse a long chain of model transformations, it should be possible to rearrange pattern rules to make the condition simple enough to understand and discharge. As a side effect, the inability to discharge or demonstrate the falsity of this condition indicates that the pattern is too complex. The poblem can be rectified by redesigning pattern with an emphasize on the use of parallel composition in the place of sequential composition, or by decomposing a pattern into sub-pattern.

Assertions There may be situations when to prove a pattern we have to assert some non-computable property of an abstract machine. For example, a pattern implementing an array sorting algorithm may need to assert that some abstract action specifies array sorting. In other words, the before-after predicate of the abstract action must be a non-deterministic statement that an array becomes sorted. There are many ways to state that an array becomes sorted and the only general solution is to prove that the action indeed does what it is expected to do. Such property is unlikely to be computable due to the size of the involved statespace, potentially infinite. Thus, we have to rely on theorem proving. For this, however, we need to know the exact input model and hence we can generate a theorem only after a pattern is instantiated.

An assertion is a simply a distinguished part of a constraining predicate. We use the following syntax to declare an assertion:

$$\textsf{assert}\ n(p)\ Q(p)$$

where n is the assertion name, p is a vector of parameters and $Q(p)$ is the property asserted. To use an assertion, one writes $n(u)$ where u is a vector of parameters.

For the example above, we assert that an action before after predicate is equivalent to our definition of a sorted array

$$\textsf{assert}\ issorted(arr, expr)\ expr \equiv \forall i \cdot (i \in 2..max(dom(arr)) \Rightarrow arr(i) \geq arr(i-1))$$

Now we can use the assertion to construct a pattern rule

```
forall $a$ where
    $\{v\} = a.\mathrm{var} \wedge issorted(v, [a])$
do
    $bubblesort(a)$
end
```

where $[a]$ is the before-after predicate of action a. For the purpose of pattern instantiation, the above is understood as

```
forall $a$ where
    $\{v\} = a.\mathrm{var} \wedge true$
do
    $bubblesort(a)$
end
```

However, for each application of $bubblesort(a)$ we get a theorem to prove: for an input model with two events each containing a single action, we get two theorems constructed from $issorted(v, a.\mathrm{exp})$ with a and v replaced by a concrete action and a concrete variable. For example, the application of the rule above to the following Event-B model

```
SYSTEM $array$
SEES $consts$
VARIABLES $ar, pos$
INVARIANT $ar : 1..n \rightarrow \mathbb{Z} \wedge pos \in 1..n$
INITIALISATION $ar := 1..n \times \{0\} \| pos := 1$
EVENTS
    $add$  =  ANY $e$ WHERE
                $e \in \mathbb{Z} \wedge pos < n$
              THEN
                $ar := ar \vartriangleleft\!\!\!\!- \{pos \mapsto n\}$
                $pos := pos + 1$
              END
    $sort$ =  ANY $e$ WHERE
                $pos = n$
              THEN
                $ar :| \forall i \cdot (i \in 1..n-1 \Rightarrow ar'(i) \leq ar'(i+1))$
              END
```

results in the following two theorems

$$ar \vartriangleleft\!\!\!\!- \{pos \mapsto n\} \equiv \forall i \cdot (i \in 1..max(dom(arr)) \Rightarrow arr(i) \geq arr(i-1))$$

$$\forall i \cdot (i \in 1..n-1 \Rightarrow ar(i) \leq ar(i+1)) \equiv \\ \forall i \cdot (i \in 2..max(dom(arr)) \Rightarrow arr(i) \geq arr(i-1))$$

Only the second theorem holds. Thus, the result of the instantiation of a pattern containing this rule is not, in a general case, a valid refinement . A more practical version of this pattern rule would ask a modeller to select an action to be refined. A modeller can make a mistake and the theorems generated from assertions should rule out the possibility of having a broken development.

The case of abstraction patterns is symmetric, with the direction of the refinement relation reversed.

We do not define what the refinement relation is and we cannot do it without narrowing the discussion to a particular formalism. The conditions and the means for demonstrating the refinement relations must be borrowed from a formalism for which the patterns mechanism is implemented.

4.5 Pattern Inverse Form

An inverse form of a pattern may be useful for a number of reasons. Firstly, many new patterns can be derived with a little effort from a collection of existing patterns. Secondly, abstraction patterns, derived from refinement patterns, may be used to redesign legacy developments by reconstructing the lost intermediate refinement steps. Thirdly, many patterns are more naturally specified as either abstraction or refinement patterns. The ability to derive mechanically the inverse of a pattern may influence the way a pattern is specified.

The inverse form of a refinement pattern is an abstraction pattern and the inverse of an abstraction pattern is a refinement pattern. While, depending on a viewpoint, the same pattern may be an inverse or direct, a pattern always remains abstraction or refinement pattern.

A pattern derived by reversing another pattern and is distinguished with the $^{-1}$ marh next to its name. An inverse of a pattern is computed by reversing its body rule:

$$\begin{array}{l}\textsf{pattern}\ p\\ \quad q\end{array}^{-1} = \begin{array}{l}\textsf{pattern}\ p^{-1}\\ \quad q^{-1}\end{array}$$

The inverse of a rule based on a model transformation is the inverse form of the model transformation, if it exists:

$$(mdr(p))^{-1} = mdr^{-1}(p)$$

Since we know that for many model transformations the inverse form does not exist, the ability to compute inverse forms of all the model transformation is the main limitation for deriving inverse patterns.

Inverse of a sequential composition is computed by applying the inverse version of the original rules in the reverse order:

$$\begin{pmatrix} q & \\ & r \end{pmatrix}^{-1} = \begin{pmatrix} r^{-1} & \\ & q^{-1} \end{pmatrix}^{-1}$$

Parallel composition is reversed by reversing its individual rules:

$$\begin{pmatrix} q \\ r \end{pmatrix}^{-1} = \begin{pmatrix} q^{-1} \\ r^{-1} \end{pmatrix}^{-1}$$

The inverse of a rule wrapped into the *forall* statement is the same statement but with the contained rule reversed:

$$\begin{pmatrix} \text{forall } a \textbf{ with } p \text{ where} \\ \quad P(a,p) \\ \text{do} \\ \quad r(a,p) \\ \text{end} \end{pmatrix}^{-1} = \begin{matrix} \text{forall } a \textbf{ with } p \text{ where} \\ \quad P(a,p) \\ \text{do} \\ \quad r^{-1}(a,p) \\ \text{end} \end{matrix}$$

There is a complication here. We cannot inverse *forall* statements with constraining predicates referencing model parts updated by the contained rule. Hence, for example, a rule removing all variables from a model -

```
forall $v$ where
    $v \in Vars$
do
    delvar($v$)
end
```

-, where transformation delvar(v) removes v from the set $Vars$, cannot be inverted since its inverse form, -

```
forall $v$ where
    $v \in Vars$
do
    addvar($v$)
end
```

- attempts to add elements from the now empty set $Vars$ and therefore does not, in a general case, undo the effect of the first rule. The condition for *forall* reversibility requires that the constraining predicate does not depend on the model parts updated by the contained rule

$$\forall s, a, p \cdot P(s, a, p) = P(\mathbf{eff}(r(a, p)), a, p)$$

where the usually implicit parameter s denotes the current model, $P(a, p) = P(s, a, p)$ and $\mathbf{eff}(s, r(a, p)$ is a model obtained by applying rule r to the current model s.

4.6 Constructing a Pattern

Many standardised design solutions (e.g. design patterns) can be used as a basis for constructing refinement and, in less extent, abstraction patterns. It is not necessarily a one-to-one mapping and there are design solutions that are so abstract that they may be not very useful when transformed into a pattern. Let us consider, as an example, the proxy design pattern ([39]) which is an abstract structuring concept for a complex system. A naive, straightforward translation of this design pattern would result in a simplistic refinement pattern for which the use of patterns and the associated proof reuse simply does not pay off. However, there can be any number of specialised versions of this design pattern formulated for particular kind of systems or specific cases. The same problem has been discovered in [83] when attempting to convert design patterns into software components.

A legacy or a current formal development is an important source of ideas for refinement automation. In many cases, concrete refinement steps are an implementation of a more general concept. Application of the concept to a whole range of possible abstract systems might result in a reusable refinement pattern. A side effect - a deeper understanding of the concept used in development through it formalisation - is important by itself.

A pattern designer is likely to have a choice of a level of a pattern presentation. A refinement pattern can be more abstract and thus be applicable to a wider range of input specifications or more specific and restricted to a narrower domain. More abstract patterns are generally easier to apply as they put less restrictions on the form of an input specification. One of the objectives during a pattern development is to find a balance between pattern generality and details in describing its functionality.

The first step of a pattern design is the identification of the possible pattern parameters, their types and restrictions. This stage defines a set of possible abstract specifications that can be transformed by this pattern and this loosely corresponds to the notion of the problem domain of a design pattern (solution). For complex patterns it is a non-trivial task to identify the weakest set of patterns requirements. The strategy is to start with a minimal set of requirements and use proof obligations as a guide to add the additional restrictions required to demonstrate pattern correctness.

It is important to carefully identify major building blocks of a pattern: new variables and events declared by the pattern, top-level matching blocks, refinements of abstract variables and events. These serve as a skeleton around which further details are added. Proofs will help to identify missing or wrong rules.

Once there is a sufficiently detailed description of a refinement pattern, proof obligations for pattern correctness can be generated. With a theorem prover, many proof obligations are discharged automatically. A proof obligation not discharged automatically may indicate a mistake in pattern but this is also a part of the pattern design routine. The structure of a proof obligation might provide valuable information on how to rectify a problem in the patterns. Typically, several iterations are needed to produce a correct pattern.

4.7 Event-B Refinement Patterns

To make pattern specifications more readable, we first introduce notational shortcuts for Event-B model transformations. Many transformations related to some model element get the element as a parameter. To avoid repetitive parameter use, a model transformation may omit a parameter implied by the transformation context.

Notational shortcuts are grouped by the context in which they are used. The same syntactic element may have different meaning in a different context. The following rules are used in the top-level context - the model context.

$\mathbf{variable}\ v$	$\mathsf{newvar}(v)$
$\mathbf{event}\ e$	$\mathsf{addevent}(e)$
$\mathbf{invariant}\ i$	$\mathsf{addinv}(i)$

Here v, e and i are constants. The rules sequentially composed with **variable** v are intepreted in the context of variable v. The same principle is applied to the event creation rule **event** e. The event scope rules have the following definition

event e

refines r	$\mathsf{refines}(e, r)$
label l	$\mathsf{newevtlabel}(e, l)$
guard g	$\mathsf{addguard}(e, g)$
param p	$\left(\begin{array}{l}\mathsf{addpar}(p)\\ \qquad \mathsf{addparam}(e, p)\end{array}\right)$
action $vsexp$	$\left(\begin{array}{l}\mathsf{newaction}(a, v, s, exp)\\ \qquad \mathsf{addaction}(e, a)\end{array}\right)$

Action, variable and parameter scope rules are simply renamed model transformations with the context element parameter omitted

action v

variable v	$\mathsf{newactionvar}(a, v)$
style s	$\mathsf{newactionsty}(a, s)$
expression e	$\mathsf{newactionexp}(a, e)$

variable v

label l	$\mathsf{varlabel}(v, l)$
action a	$\mathsf{newvaraction}(v, a)$
invariant t	$\mathsf{newvartype}(v, t)$

param v

label l	$\mathsf{parlabel}(a, l)$
invariant t	$\mathsf{newpartype}(a, t)$

In case there is no explicit declaration of a context, suffix **for** b is used to point at a context element. Thus, **action** a **for** $somevar$ is understood as

$$\mathsf{newvaraction}(somevar, a)$$

The additional benefit of rule context is that many well-formedness conditions are eliminated. For example, the fact that **style** s appears in a context of an action, makes it redundant to prove (when analysing a pattern correctness) that parameter a of $\mathsf{newactionsty}(a, s)$ is an existing action. The absence of any rules removing model elements means that the conflict-freeness conditions can be relaxed.

If an element is declared using the shortcut notation but is not given a name, we assume that a suitable fresh name is provided during pattern instantiation. Otherwise, the labels for all the new named elements must be declared as rule or pattern parameters.

For example, the following snippet of a refinement rule

```
newvar($v$)
  varlabel($v, l$)
    newvaraction($v, a$)
      newvartype($v, t$)
```

with the shortcut notation is represented as

```
**variable** $v$
  **label** $l$
  **action** $a$
  **invariant** $t$
```

Unless l is substituted with a constant label, the above can be written as

```
**variable** $v$
  **action** $a$
  **invariant** $t$
```

4.8 Messaging Pattern

This section illustrates the patterns mechanism with an example of an Event-B refinement pattern. The presented pattern transforms an input model by refining an action of an abstract event into a simple communication mechanism. The complete pattern specification is given in Figure 4.2.

The pattern is applicable to models with at least one event that contains an action assigning to a single variable (in a general case, an Event-B action updates a vector of variables). This condition is expressed using the *with* construct which also defines the parameters $destevt$ (some abstract event) and $copyact$ (some action of $destevt$) (replacing *with* with *forall* we could make the pattern transform all the suitable pairs of events and actions in an input model). The pattern body is made of five major blocks: the construction of a variable representing a communication channel (block ch in Figure 4.2); the construction of a channel state variable (block rd); the defintion of a new event responsible for writing in the channel ($send$); the refinement rules updating the abstract event $destevt$; the invariant relating the state of the channel variable ch with the abstract model state (the last rule in Figure 4.2). Notation $[var\ style\ expr]$ used in the invariant rule stands for a before-after predicate corresponding to the Event-B action $var\ style\ expr$.

```
pattern $messaging$
  forall $vv$ with $destevt, copyact$ where
      $destevt \in evt \wedge copyact \in destevt.\text{actions} \wedge copyact.\text{var} = \{vv\}$
  do
      $ch$ : [ variable $ch$
                 invariant $ch \in vv.\text{type}$
                 action $ch :\in vv.\text{type}$ ]
          $rd$ : [ variable $rd$
                     label $ch$_rd
                     invariant $rd \in \mathbb{B}$
                     action $rd := false$ ]
              $send$ : [ event $send$
                           param $destevt.\text{arg}$
                           guard $destevt.\text{guard}$
                           guard $rd = false$
                           action $ch := copyact.\text{expression}$
                           action $rd := true$ ]
              $recv$ : [ guard $rd = true$ for $destevt$
                           action $rd := false$ for $destevt$
                             expression $ch$ for $copyact$ ]
              invariant $rd = true \Rightarrow [ch\ copyact.\text{style}\ copyact.\text{expression}]$
  end
```

Figure 4.2: The *messaging* refinement pattern specification.

4.8.1 Correctness

To demonstrate the pattern correctness, we routinely apply the conditions from Section 4.4.2 to obtain the list of the pattern proof obligations. Instantiating the computability proof obligation, we get the following condition:

$finite(\{(d, c) \mid d \in evt \wedge c \in d.\text{actions} \wedge card(c.\text{var}) = 1\})$

The above trivially holds noticing that an Event-B model has a finite number of events - $finite(evt)$ - and an event has a finite number of actions: $finite(d.\text{actions})$.

The conflict-freeness condition is demonstrated by writing out scopes of the pattern rules. For example, for the pattern subset declaring new variable ch the scopes are computed as follows

variable ch	$var, \star ch$
invariant $ch \in copyact.\text{type}$	$\star ch : \text{typ}$
action $ch :\in copyact.\text{type}$	$\star ch : \text{act}$
variable rd	$var, \star rd$

For all the pairs of parallel rules we have to demonstrate the absence of scope conflicts. Although this results in a large number of proof obligations, all such obligations are computed automatically by a tool. Moreover, in the new version of the tool it is impossible to construct patterns with scope conflicts: the editor simply does not allow parallel composition of conflicting rules.

Well-formedness of a pattern is analysed by generating corresponding proof obligations for all the pattern rules. For our example, there are 17 such proof obligation. The declaration of new variable ch results in the following proof obligations

invariant preservation: $ch' \in vv.\mathrm{type} \Rightarrow ch' \in vv.\mathrm{type}$

feasibility: $\mathcal{I}(v) \Rightarrow \exists ch' \cdot ch' \in vv.\mathrm{type}$

The names on the left-hand side indicate the Event-B refinement conditions [20] from which the proof obligations are derived. Note that $vv.\mathrm{type}$ is some unknown predicate. While the first proof obligation is trivially true, to discharge the second one we have to rely on the properties of the abstract model being transformed by the pattern. This properties are summarised in predicate $\mathcal{I}(v)$, introduced later in this section. For brevity, we present only few interesting proof obligations.

The **guard** $rd = true$ **for** $destevt$ rule leads to a proof obligation requiring that the overall guard of concrete event $destevt$ is stronger than the guard of its abstraction. This condition trivially holds since the new guard is stronger due to the conjunction with an additional condition:

guard strengthening: $destevt.\mathrm{guard}(v) \land rd = true \Rightarrow destevt.\mathrm{guard}(v)$

The update of the expression of the $copyact$ (rule **expression** ch **for** $copyact$) leads to two proof obligations. The first requires us to show that the new action is feasible: always delivers a result under the given assumptions. The second states that new action must be a refinement of the corresponding action of abstract event $destevt$:

feasibility: $\mathcal{I}(v) \land J(v,u) \land destevt.\mathrm{guard}(v) \land rd = true \Rightarrow$
$\exists u' \cdot [copyact.\mathrm{var} \ := \ ch](u,u')$

refinement: $\mathcal{I}(v) \land J(v,u) \land destevt.\mathrm{guard}(v) \land rd = true \land$
$[copyact.\mathrm{var} \ := \ ch](u,u') \Rightarrow$
$\exists v' \cdot ([copyact](v,v') \land J(v',u'))$

where v is a vector of abstract model variables. This vector contains at least one variable vv, $\{vv\} = copyact.\mathrm{var}$. Vector u denotes concrete model variables, which, among the variables inherited from an abstract model ($v \subset u$), contains variables ch and rd.

Predicate $\mathcal{I}(v)$, used in several proof obligations above, is a collection of known facts about the abstract model. For the messaging pattern, we know that a suitable abstract model contains at least one event which must have at least one action assigning to a variable. We also assume that an abstract model transformed by the pattern is well-formed (in sense of Event-B well-formedness [20]). Thus we may state that the abstract action $copyact$ has a solution when its guard is enabled (the feasibility condition). This condition helps us to discharge the *feasibility* proof obligation above. The complete $\mathcal{I}(v)$ definition is

$$\begin{aligned}\mathcal{I}(v) \quad = \quad & I(v) \wedge \exists e \cdot (e \in evt \wedge \exists a \cdot (a \in e.\text{act} \wedge card(a.\text{var}) = 1)) \wedge \\ & (I(v) \wedge destevt.\text{guard}(v) \Rightarrow \exists v' \cdot ([copyact](v, v') \wedge J(v', u')))\end{aligned}$$

where $I(v)$ is an arbitrary predicate corresponding to the abstract model invariant.

Predicate $J(v, u)$ is a concrete invariant. In addition to constraints on concrete variables, a concrete invariant in Event-B also has a gluing invariant linking the states of concrete and abstract model. In our case, such gluing invarint is the invariant produced by the rule **invariant** $rd = \ldots$. This invariant links the abstract action expression with the value of concrete variable ch. The complete definition of $J(v, u)$ is as follows:

$$\begin{aligned}J(v, u) \quad = \quad & ch :\in copyact.\text{type}(v) \wedge rd \in \mathbb{B} \wedge \\ & rd = true \Rightarrow [ch\ copyact.\text{style}\ copyact.\text{expression}](u)\end{aligned}$$

The *feasibility* and *refinement* conditions are discharged in a number of simple steps by writing out $\mathcal{I}(v)$ and $J(v, u)$ and then demonstrating that

$$[copyact.\text{var} \ := \ ch](u, u')$$

is equivalent to $[copyact](v, v')$ provided $rd = true$.

Several model transformations require further analysis to ensure that the pattern construct a model refinement. These rules are **invariant** $rd = \ldots$, **variable** $destevt.\text{arg}$ and **event** $send$. The former two cases are rather trivial and we only consider the condition generated for the **event** $send$ rule. This condition requires us to prove that new event $send$ terminates in a finite number of steps and passes the control to some other event. For this we have to find a variant which is a finite natural number decremented by the new event:

non-divergence: $$\begin{aligned}& \mathcal{I}(v) \wedge J(v, u) \wedge destevt.\text{guard}(v) \wedge rd = false \wedge \\ & [\{ch\}\ copyact.\text{style}\ copyact.\text{expression}](ch, ch') \wedge \\ & rd' = true \Rightarrow \exists V \cdot (V(u) \in \mathbb{N} \wedge V(u') < V(u))\end{aligned}$$

One possible witness for V is $V(rd), V : \mathbb{B} \rightarrow \{0, 1\} = \{true \mapsto 1\} \cup \{false \mapsto 0\}$. Event $send$ decreases V: in the before-state state of $send$, V must be 1 and in the after-state it is always 0.

Figure 4.3 demonstrates an example of the pattern instantiation. The input model contains a single event with two actions. Any of these actions can be selected as a value for the $copyact$ parameter and it is assumed that the parameter selection is handled externally (e.g., by a modeller). In this example the pattern is instantiated with $copyact = (n := (m + 1)\ mod\ 6)$. In Figure 4.3, model $m0$ is the input model and model $m1mes$ is the the model produced by the messaging refinement pattern (Figure 4.2).

SYSTEM $m0$
VARIABLES n, m
INVARIANT $n \in 0\,..\,5 \wedge m \in \{1,2,3\}$
INITIALISATION $n := 0 \| m := 1$
EVENTS
$\quad work \quad = \quad$ **BEGIN** $n := (m+1)\ mod\ 6 \| m :\in \{1,2\}$ **END**

SYSTEM $m1mes$
REFINES $m0$
VARIABLES n, m, ch, ch_rd
INVARIANT
$\quad n \in 0\,..\,5 \wedge m \in \{1,2,3\} \wedge ch \in 0\,..\,5 \wedge ch\_rd \in \mathbb{B}$
$\quad ch\_rd = true \Rightarrow ch = (m+1)\ mod\ 6$
INITIALISATION $n := 0 \| m := 1 \| ch :\in 0\,..\,5 \| ch\_rd := false$
EVENTS
$\quad send \quad = \quad$ **WHEN** $ch\_rd = false$ **THEN** $ch := (m+1)\ mod\ 6 \| ch\_rd := true$ **END**
$\quad work \quad = \quad$ **WHEN** $ch\_rd = true$ **THEN** $n := ch \| m :\in \{1,2\} \| ch\_rd := false$ **END**

Figure 4.3: Instantiation example for the *messaging* refinement pattern.

The *messaging* pattern is applicable to a wide range of models. One example is a combination with the recovery block [81], parity bit [63] and hamming code [63] patterns to produce a fault-tolerant communication protocol. The protocol starts with a communication loop with a single parity check bit. Upon detection of an error, the protocol switches to a communication loop using the hamming code. The described development can be constructed entirely from the currently available refinement patterns.

4.9 Recovery Block Pattern

This pattern helps to develop software capable of tolerating software faults by introducing N alternatives designed diversely following the ideas from [62]. Checkpointing is used to save the state before executing an alternative so that results of unsuccessful execution can be discarded. An alternative execution is followed by checking an acceptance test. If the test is passed then the result of the current alternative is used as the final result. Otherwise, the state is rolled back and another alternative is activated (Figure 4.4). If no alternate is available, an exception is propagated.

The pattern takes as input a model with two events. One of the events is the specification of a desired behaviour. The other event is the connection to some external recovery or abortion mechanism. During instantiation, the pattern also asks for the number N of behaviour block instances.

Further refinements should diversify designs of behaviour alternatives (e.g. by enforcing the use of different solutions and by involving different developers) and adapt test conditions. A good starting point for applying this pattern is a specification with non-deterministic before-after predicates. The conjunction of all before-after predicates of an abstract behaviour event is used by the pattern as the acceptance test. The pattern has two parameters - an abstract event and a value representing the number of blocks in the refined model:

```
pattern $recblock$
   req_$typing$ $b \in evt \wedge n \in finite(\mathbb{N})$
   req_$notempty$ $card(b.\mathrm{act}) > 0$
   req_$notzero$ $n > 0$
```

Here b is an abstract event specifying the desired system behaviour and n is a number of recovery blocks. The pattern requirements state the typing of the parameters and the fact that the behaviour event contains at least one action andthe number of blocks is not zero.

This pattern can be applied to any specification with at least one event which must be not empty (contain actions). The pattern makes no additional assumptions about event bodies, guards and parameters as the pattern is general enough to handle all the possible cases.

The pattern introduces two new variables (these variables appear in the output specification) to model control flow for the new events. Variable br defines the currently active behaviour block. When its value goes beyond the allowed block index, it indicates failure of all the blocks. Variable st indicates the current stage: checkpoint ($st = 0$), block ($st = 1$) or acceptance test ($st = 2$):

$$ctrvar = \left(\begin{array}{l} \textbf{variable } br \\ \quad \textbf{invariant } br \in 0..(n+1) \\ \quad \textbf{action } br := 0 \\ \textbf{variable } st \\ \quad \textbf{invariant } st \in 0..2 \\ \quad \textbf{action } st := 0 \end{array} \right)$$

The pattern models checkpointing by extending system state with new variables used to hold intermediate results produced by the alternatives. If the result of an alternative fails the acceptance test, the state extension is disregarded. When test succeeds, the state is used as the final result. This approach allows us to intro-

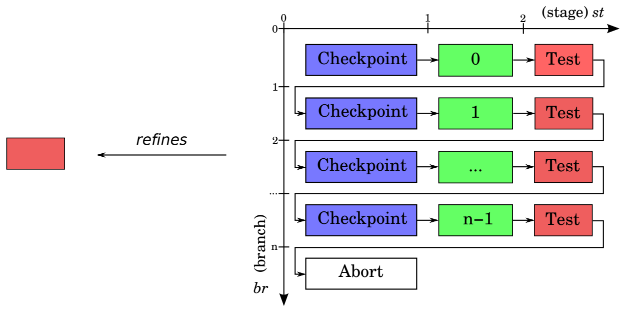


Figure 4.4: The Recovery Block pattern. The checkpoint and alternatives are modelled as new events, the *Test* block refines the behaviour abstract event.

duce checkpoints without knowing the whole system state. The following pattern fragment creates a copy of each variable assigned in the event b:

```
shdvar = (
  forall $a, av$ where
      $a \in b.\mathrm{act} \wedge a.\mathrm{var} = \{av\}$
  do
      variable $cpvar$
         label cp_$av$.nom
         invariant $cpvar \in av.\mathrm{typ}$
         action $cpvar\ av.\mathrm{act.sty}\ av.\mathrm{act.exp}$
  end
)
```

This pattern rule creates a new variables for each variables assigned in the abstract event b. Local variable a is used to iterate over the actions of event b while av binds to a variable assigned by action a. The body of the loop construct is a typical rule set for a new variable declaration.

The pattern fragment below creates a checkpoint event which saves the current values of the variables updated in the event b. This event is enabled when $st = 0$:

```
chkptevt = (
  event $chkpt$
     label $b$_chkpt
     guard $st = 0$
     forall $a, av$ where
         $a \in b.\mathrm{act} \wedge a.\mathrm{var} = \{av\}$
     do
         action cp_$av$.nom $:= av.\mathrm{nom}$
     end
     action $st := 1$
)
```

The event advances the stage variable st so that a currently selected alternative is enabled. An alternative contains the same set of actions as the abstract event b.

These actions assign to the copies of the abstract variables updated in b. Although, an alternative is formally not a refinement of b, it is related through the actions. A designer has the choice of changing alternatives behaviour just after applying the pattern or keeping them intact and using refinement to gradually introduce specialisation. In the latter case, the actions derived from the actions of b serve as an abstract specification for further refinements. To allow for meaningful refinements these actions must be non-deterministic. The next pattern fragment produces n events representing recovery block alternatives:

```
altevt = (
forall $i$ where
    $i \in 1..n$
do
    event $alt_i$
        label $b$_alt_$i$
        guard $st = 1$
        guard $br = i - 1$
        guard $b$.grd
        variable $b$.var
        $actions$    action $st := 2$
end
)
```

where rule $actions$ is defined as

```
forall $a, av$ where
    $a \in b.\mathrm{act} \wedge a.\mathrm{var} = \{av\}$
do
    action cp_$av$.nom $a$.sty $a$.exp
end
```

Note the guard $br = i - 1$ selecting the current alternative and action $st := 2$ enabling the acceptance test. The acceptance test event checks if the alternative has succeeded and, if it is so, uses its result as the final result. The acceptance test must refine b since it is the only event which is allowed to update inherited abstract variables which the abstract version of b used to produce the result. In other words, an input specification is transformed in such a way that parts which the pattern is not aware about are not effected.

The acceptance test is computed automatically by the pattern from the abstract event b. In English, the acceptance can be informally formulated as *"any result that agrees with the specification of the abstract event b is acceptable"*. To give the exact meaning to the *"agrees with"* phrase we use the before-after predicates of the abstract event b:

$$accgrd = \left(\begin{array}{l} \textbf{guard}\ st = 2\ \textbf{for}\ b \\ \textsf{forall}\ a, av\ \textsf{where} \\ \quad a \in b.\mathrm{actions} \wedge a.\mathrm{var} = \{av\} \\ \textsf{do} \\ \quad \textbf{guard}\ [\mathtt{cp\_}av.\mathrm{nom}\ av.\mathrm{sty}\ av.\mathrm{exp}]\ \textbf{for}\ b \\ \quad \textbf{action}\ av.\mathrm{nom} := \mathtt{cp\_}av.\mathrm{nom}\ \textbf{for}\ b \\ \textsf{end} \end{array}\right)$$

where $[var\ style\ exp]$ is a before-after predicate correcponding to the substitution $var\ style\ exp$.

We also have to address the case when the acceptance test fails. For this we declare a new event and use a guard which is the opposite of the acceptance test rule. One of the responsibilities of this event is to advance the br variable so that a new alternative is used next time:

$$testevt = \left(\begin{array}{l} \textbf{event}\ test\_fail \\ \quad \textbf{label}\ b\mathtt{\_test\_fail} \\ \quad \textbf{guard}\ st = 2 \\ \quad \textbf{guard}\ \bigvee_{a \in b.\mathrm{act} \wedge a.\mathrm{var} = \{av\}} \neg\, [\mathtt{cp\_}av.\mathrm{nom}\ a.\mathrm{sty}\ a.\mathrm{exp}] \\ \quad \textbf{action}\ br := br + 1 \\ \quad \textbf{action}\ st := 0 \end{array}\right)$$

Since we have only n alternatives with indices $0..n-1$, a state where $br = n$ indicates that all the alternatives have failed to produce an acceptable result. To cover the case of $br = n$ the pattern produces a new events which simply uses the abstract event b behaviour to produce some ”safe” result:

$$failevt = \left(\begin{array}{l} \textbf{event}\ fail \\ \quad \textbf{label}\ b\mathtt{\_fail} \\ \quad \textbf{refines}\ b \\ \quad \textbf{guard}\ br = n \\ \quad \textbf{guard}\ b.\mathrm{grd} \\ \quad \textbf{param}\ b.\mathrm{var} \\ \quad \textbf{action}\ b.\mathrm{act} \end{array}\right)$$

Finally, the complete pattern is

```
pattern $recblock$

  req_$typing$ $b \in \mathbf{Event} \wedge n \in finite(\mathbb{N})$
  req_$notempty$ $card(b.\mathrm{act}) > 0$
  req_$notzero$ $n > 0$
    $ctrvar$
      $shdvar$
        $chkptevt$
        $altevt$
        $testevt$
        $failevt$
        $accgrd$
```

In Appendix B, this pattern is given in the notation accepted by a tool working with Event-B patterns. We discuss the tool in Section 6.6. The pattern as it is given in Appendix B can be used as it is by anyone interested in modelling recovery block with the Event-B method. The pattern instantiation process is interactive and self-explanatory. Also, a modeller would not normally have to look into a pattern source to apply it.

We continue the pattern discussion with the analysis of the pattern correctnes. For this pattern we are able to demonstrate the once-for-all correctness.

4.9.1 Recovery Block Pattern Correctness

In this section we demonstrate that the recovery block pattern indeed produces valid refinements for any input specification to which it can be applied. Here we write out and analyse proof obligations manually. Most of this can be handled by a tool and we are working on adding support for generating proof obligations and automatically discharging them with the platform theorem prover.

Computability Computability is trivially satisfied since an Event-B model has a finite number of elements and parameter n is finite by declaration:

$$\forall e \cdot (e \in \mathrm{EVENT} \Rightarrow finite(e.\mathrm{act})) \wedge finite(n)$$

Conflict-freeness We have to demonstrate for the following pattern rule, based on the parallel composition, that there are no rules with overlapping scopes:

$$\begin{array}{l} chkptevt \\ altevt \\ testevt \\ failevt \\ accgrd \end{array}$$

The individual rule scopes are

$$\begin{array}{lcl}
scope(chkptevt) & = & \star chkpt, \star chkpt\!:\!\mathrm{act}, \star chkpt\!:\!\mathrm{nom}, chkpt\!:\!\mathrm{grd} \\
scope(altevt) & = & \star alt_i, \star alt\!:\!\mathrm{act}, \star alt\!:\!\mathrm{nom}, alt\!:\!\mathrm{grd} \\
scope(accgrd) & = & b\!:\!\mathrm{grd}, \star b\!:\!\mathrm{act} \\
scope(test\_event) & = & \star test\_fail, \star test\_fail\!:\!\mathrm{nom}, \star test\_fail\!:\!\mathrm{act}, test\_fail\!:\!\mathrm{grd} \\
scope(failevt) & = & \star failevt, \star failevt\!:\!\mathrm{nom}, \star failevt\!:\!\mathrm{ref}, \star failevt\!:\!\mathrm{act}, failevt\!:\!\mathrm{grd}
\end{array}$$

where $i \in 1..n$. The rule scopes are clearly disjoint.

Well-formedness Most of the proofs fall into the well-formedness category.

Declarations of br and st result in the following proof obligations:

$$\begin{array}{ll}
\mathsf{PAT\_FIS\_INI}_{br} & \exists br' \cdot (br \in 0..(n+1) \wedge br' = 0) \\
\mathsf{PAT\_INV\_INI}_{br} & br \in 0..(n+1) \wedge br' = 0) \Rightarrow br' \in 0..(n+1) \\
\mathsf{PAT\_FIS\_INI}_{st} & \exists st' \cdot (st \in 0..2 \wedge st' = 0) \\
\mathsf{PAT\_INV\_INI}_{st} & st \in 0..2 \wedge st' = 0 \Rightarrow st' \in 0..2
\end{array}$$

which are trivially true. Note that this conditions are devised by instantiating the pattern correctness conditions defined in Section 4.4.2 with Event-B model transformations.

The pattern introduces new system variables supporting checkpointing. For each variable updated in the event b a new variable is created with the same type and initial state. To express this, the pattern uses the **forall** construct. Consequently, resulting proof obligations use the universal quantifier

$$\begin{array}{ll}
\mathsf{PAT\_FIS\_INI}_{cpvar} & \forall a \cdot (SimpleAct(a, b) \Rightarrow \exists c' \cdot (c' \in Tp(a) \wedge [c'\ St(a)\ In(a)])) \\
\mathsf{PAT\_INV\_INI}_{cpvar} & \forall a \cdot (SimpleAct(a, b) \Rightarrow [c\ St(a)\ In(a)] \Rightarrow c' \in Tp(a))
\end{array}$$

where $[vse]$ is a before-after predicate of an action made from variable v, action style s (:=, :∈ and $\mid\ \in$) and expression s. Also, the following shortcuts are used: $SimpleAct(a, b) \overset{\mathrm{df}}{=} a \in b.\mathrm{act} \wedge a.\mathrm{var} = \{av\}$, $c = cpvar$, $St(a) = av.\mathrm{act.sty}$, $Tp(a) = av.\mathrm{typ}$ and $In(a) = av.\mathrm{init.exp}$. The proof obligations above are simplified by removing quantifier $\forall a$. To do the proof we use the information about the abstract variables from which the copied variables are derived:

PAT_FIS_INI$_{cpvar}$ $\quad SimpleAct(a,b) \Rightarrow \exists v' \cdot (v \in Tp(a) \wedge [v'\ St(a)\ In(a)]) \vdash$
$\quad\quad SimpleAct(a,b) \Rightarrow \exists c' \cdot (c' \in Tp(a) \wedge [c'\ St(a)\ In(a)])$

PAT_INV_INI$_{cpvar}$ $\quad SimpleAct(a,b) \Rightarrow [v\ St(a)\ In(a)] \Rightarrow v' \in Tp(a)) \vdash$
$\quad\quad SimpleAct(a,b) \Rightarrow [c\ St(a)\ In(a)] \Rightarrow c' \in Tp(a))$

These conditions are trivially correct as the left and right parts differ only in the names of free variables. Note, that the proof covers the general case of creating variable copies.

The checkpoint event initialises sub-states used by the recovery blocks. The action updating st gives rise to the following trivial proof obligations:

PAT_REF_FIS$_{chkpt2}$ $\quad I(v) \wedge st \in 0..2 \wedge st = 0 \Rightarrow \exists st' \cdot (st \in 0..2 \wedge st' = 1)$

PAT_NEW_INV$_{chkpt2}$ $\quad I(v) \wedge st \in 0..2 \wedge st = 0 \wedge st' = 1 \Rightarrow st' \in 0..2$

Initialisation of checkpoint variables uses the *forall* statement and hence the universal quantifier appears in the proof obligations:

PAT_REF_FIS$_{chkpt1}$ $\quad \forall a \cdot (SimpleAct(a,b) \Rightarrow$
$\quad\quad (I(v) \wedge c(a) \in Tp(a) \wedge v(a) \in Tp(a) \wedge st = 0 \Rightarrow$
$\quad\quad\quad \exists c(a)' \cdot (c(a) \in Tp(a) \wedge c'(a) = var(a))))$

PAT_NEW_INV$_{chkpt1}$ $\quad \forall a \cdot (SimpleAct(a,b) \Rightarrow$
$\quad\quad (I(v) \wedge c(a) \in Tp(a) \wedge v(a) \in Tp(a) \wedge st = 0 \wedge$
$\quad\quad\quad c'(a) = var(a) \Rightarrow c'(a) \in Tp(a)))$

where $c(a) = \texttt{cp\_}av.\mathrm{name}$, $v(a) = av.\mathrm{name}$ and $St(a)$, $In(a)$ and $Tp(a)$ as defined above and name av is bound by predicate $SimpleAct$. The quantifier can be dropped and with the addition properties of the original variables as hypothesis these conditions are demonstrated in a manner similar to the above.

The pattern fragment creating the recovery blocks employs two *forall* statements. The outer one runs through all the recovery block indices and the inner one creates a new action for each action in the abstract event b. The proof obligations are as follows:

PAT_REF_FIS$_{alt}$ $\quad \forall i \cdot (i \in 1..n \Rightarrow \forall a \cdot (SimpleAct(a,b) \Rightarrow$
$\quad\quad (I(v) \wedge c \in Tp(a) \wedge st = 1 \wedge br = i - 1 \Rightarrow$
$\quad\quad\quad \exists c' \cdot (c' \in Tp(a) \wedge [c'\ St(a)\ Ex(a)]))))$

PAT_NEW_INV$_{alt}$ $\quad \forall i \cdot (i \in 1..n \Rightarrow \forall a \cdot (SimpleAct(a,b) \Rightarrow$
$\quad\quad (I(v) \wedge c \in Tp(a) \wedge st = 1 \wedge br = i - 1 \wedge [c\ St(a)\ Ex(a)] \Rightarrow$
$\quad\quad\quad c' \in Tp(a))))$

These are easy to demonstrate for a case of some index i and some action a. It is also known that the abstract actions are well-formed and this is used as a hypothesis. The case for the action assigning to st is trivial.

The acceptance test defines actions replacing the abstract actions of event b. We have to prove that under the given conditions each such action refines its abstract counterpart:

$$\begin{array}{ll} \mathsf{PAT\_REF\_FIS} & \forall a \cdot (SimpleAct(a,b) \Rightarrow \\ & \quad (Tp(a) \Rightarrow \exists v'(a) \cdot (c(a) \in Tp(a) \land v'(a) = c(a)))) \\ \mathsf{PAT\_REF\_INV} & \forall a \cdot (SimpleAct(a,b) \Rightarrow \\ & \quad (I(v) \land [c(a)\ St(a)\ Ex(a)] \land v(a) = c(a) \Rightarrow \\ & \quad\quad \exists v'(a) \cdot ([v'(a)\ St(a)\ Ex(a)] \land va(a) \in Tp(a)))) \end{array}$$

These conditions are trivially correct. For concrete version of event b we have to demonstrate that the new guard is stronger than its abstract counterpart. It is indeed so, as the pattern fragment strengthens the guard with the additional conditions. For the new event $test$ we have several trivial proof obligations due to the actions $br := br + 1$ and $st := 0$.

Refinement Omitting a multitude of trivial refinement conditions from non-refinement model transformations, the only non-trivial refinement condition is the non-divergence of new events constructed by the pattern.

To prove the non-divergence, we have to demonstrate that there exists such $V \in \mathbb{N}$ that it is decreased by all the new events:

$$\begin{array}{ll} \mathsf{PAT\_NEW\_DIV} & I(v) \land J(v,w) \land H(w) \land S(w,w') \Rightarrow \\ & \quad V(w) \in \mathbb{N} \land V(w') < V(w) \end{array}$$

Let $V = (n+1)*3+2-(br*3+st)$ and $T = st \in 0..2 \land br \in 0..(n+1) \land n \in \mathbb{N} \land n > 0$

Condition $T \Rightarrow V(w) \in \mathbb{N}$ holds since $\max(br*3+st) = (n+1)*3+2$. To prove that all the events decrease V we have to demonstrate that the following conditions hold

$$\begin{array}{ll} \mathsf{PAT\_NEW\_DIV}_{chkpt} & T \land st = 0 \land st' = 1 \land br' = br \Rightarrow \\ & \quad V(st', br') < V(st, br) \\ \mathsf{PAT\_NEW\_DIV}_{alt} & \forall i \cdot (i \in 1..n \Rightarrow \\ & \quad (T \land st = 1 \land st' = 2 \land br' = br \Rightarrow V(st', br') < V(st, br))) \\ \mathsf{PAT\_NEW\_DIV}_{test} & T \land st = 2 \land st' = 0 \land br' = br + 1 \Rightarrow V(st', br') < V(st, br) \end{array}$$

The first two expressions increment st leaving br unchanged and hence decrease the variant expression since V is a monotonously decreasing function. The $test$ event resets st to zero but this is compensated for by the increment of br.

This pattern and the related N-versioning programming pattern (Section 6.2) have been applied in the development of the Ambient Campus case study within the ICT RODIN Project [84]. In this case study we have developed several application scenarios for PDAs and smartdust devices in which fault tolerance is essential just to achieve a reasonable usability level. In particular, we have used the Recovery Block pattern to alternate between different positioning services: GPS (fails indoors), motes (fails when there are not enough motes in proximity) and the WiFi base-station association.

4.10 Using Patterns

Our experience suggests that the refinement pattern mechanism can make a considerable impact on the development process and make a formal development process more economical.

One of the attractive features of the refinement patterns is that pattern application supported by the right tool is almost instantaneous and straightforward. Different refinement paths can be investigated without investing considerable time and modelling efforts by just selecting different patterns. This presents a considerable advantage over the manual refinement where a developer could be reluctant to redo modelling steps once committed to a particular solution.

Reading and applying refinement patterns is much easier than writing them. Hence, the mechanism of patterns makes it possible to differentiate between the roles of a formal method expert, designing high-quality reusable patterns, and an engineer, using patterns to design a system model. The fact that application of patterns does not require a high level of expertise in formal methods can contribute to the wider adoption of formal modelling as a cost-effective software engineering technique for safety-critical and dependable systems.

The effect the pattern mechanism can have on the development process depends on the number and the quality of available patterns. We consider pattern correctness as an important aspect of pattern quality: though patterns not producing any useful transformations are correct, there can be no patterns constructing invalid refinements.

Unlike concrete refinement steps, patterns are designed to be reusable. An important part of the pattern mechanism is a facility to look for a pattern implementing some specific transformations.

A refinement pattern is a complete, self-sufficient unit that can be communicated between developers to support reuse and experience sharing. With an extensive pattern library, a whole system design can be realised as a composition of third-party patterns with some custom logic filled in places.

4.11 Pattern Classes

In a general case, by a refinement or abstraction pattern we understand a collection of rules that comprise all kinds of model transformation rules: deletion, replacement of model elements and addition of new elements. For many patterns, however, it is sufficient to use a limited subset of available model transformations. Such patterns may be easier to construct, analyse and apply.

4.11.1 Superposition Pattern

A superposition pattern neither changes nor removes existing model elements and only adds new model elements, in other words, superimposes new behaviour onto the abstract behaviour [35, 85]. A more notable example of an approach working with this class of patterns is the CIP project transformational language, CIP-S [86].

The Recovery Block pattern discussed in Section 4.9 is an example of a superposition refinement pattern.

An attractive property of a superposition pattern is the ability to unambiguously describe such pattern using only input and output model classes. Hence, if we know that the pattern in question is a superposition refinement pattern it is enough to exhibit its input and output model classes and the pattern itself can be trivially derived.

For example, a simple pattern, adding a new a new empty empty at a specific point of a model, is fully characterised by its input and output model classes and can be defined as follows

```
pattern seq
```

```
FACET incl(seq)
EVENTS
    ?e  =  WHEN ?g THEN skip END
```

```
FACET outcl(seq)
REFINES incl(seq)
VARIABLES ?v
INVARIANT ?v ∈ 𝔹
INITIALISATION ?v := false
EVENTS
    ?ne  =  WHEN ?g∧?v = false THEN ?v := true END
    ?e   =  WHEN ?g∧?v = true THEN ?v := false END
```

4.11.2 Mapping Pattern

The class of mapping patterns is a subclass of superposition patterns. These patterns only insert new top-level model elements. A pattern from this class, for example, can add a new event but cannot add a new action or a guard to an existing event.

The TLA [8] pattern integration mechanism [87] uses this kind of patterns.

4.11.3 (De-)Integration Pattern

An integration pattern is a special case of a mapping pattern. Patterns of this class put no restrictions on the input model class. This pattern type can be understood as a composition of two models - the input model and the model contained in a pattern. This kind of patterns were used (with a manual instantiation) in the classical B Method case study - the Mechanical Press Controller [88]. Since this pattern class does not restrict the input model class, an integration pattern can be described by the output model class alone.

The abstraction case of this pattern class is called de-integration pattern.

4.11.4 Presentation Pattern

A presentation pattern constructs a refinement (or abstraction) that can be refined (abstracted) again to the initial model. The simplest case of such pattern is a pattern renaming a model element (and consistently updating all other model elements). A more interesting case is a data refinement that can be undone without losing any details on the behaviour of the modelled system. In Section 6.4 we describe in details a pattern from this class which transforms a set variable into a characteristic function of a set.

4.12 Summary

The pattern mechanism is a step towards making formal methods more accessible and less expensive. Pattern is a program rewriting a formal model to construct a new model. Unlike conventional programs, created with mainstream programming languages, the patterns are designed to make it possible to demonstrate pattern correctness mechanically, using automated theorem provers, e.g. [89, 90]. Correctness proofs, once constructed for a pattern, automatically guarantee correctness of any possible pattern instantiation.

Patterns are of a great assistance in modelling and, we believe, can be effectively used by non-experts. Pattern instantiation process is interactive: the existing tool prototype dynamically creates a pattern instantiation wizard which guades a modeller through the process of pattern application. The example we have considered in this chapter was developed using the tool and is available for download from the tool web-page [63].

In the next chapter we discuss how to describe a whole development as a composition of refinement patterns or, which is the same, a single refinement pattern.

Chapter 5

Modelling with Patterns

5.1 Introduction

A model is a mathematical description of a system to be constructed. A detailed system description is made in several steps by formally transforming an initial model until the result becomes sufficiently detailed (e.g., can be translated into a programming language). Modelling is a complex activity. While, from the mathematical viewpoint, formal step-wise development is a sequence of formal modifications to an initial abstract model, for a modeller it is a mental elaboration of a system design; the mathematical notation and proofs assist in this process. Similarly, a model has two interpretations - one is given by the semantics of the chosen formal method and another, probably more important one, exists in a modeller's head and, possibly, is described somewhere informally. At some point, mental and formal models become close enough so that a modeller sees no difference and states that the formal model is an accurate reproduction of the informal mental model. If a modeller's concept of the modelled system is hazy, it is very unlikely that the resulting formal model is accurate and adequate despite being mathematically correct. Requirements engineering [91] helps to elaborate informal system model prior to the formal modelling stage. Ideally, given a formal model and its requirements one should be able to state whether the model is a satisfactory implementation of the requirements. Even with a perfect requirements document there is still a problem of capturing all the system details in a formal model. Intuitively, to construct a formal model with a step-wise refinement procedure, each refinement step should be taking the model closer to the modelling goal corresponding to the system requirements. The refinement method makes no claims to be able do that. The problem can be related to finding a path between two points. If from the first point the other can be seen, it is possible just to walk straight towards the second point. But if the second point cannot be seen, a lot of effort may be wasted trying to find it and it may be never reached at all. To construct a refinement path from an abstraction towards and an implementation, a modeller must gets some guidance from a tool. Such guidance would assist a modeller in constructing intermediate models and enusre that the modelling goal is eventually achieved.

Construction of a complex system is aided by one or more problem-specific methods, such as software engineering methods. For example, a space probe controller would follow methods elaborated for development of space probe software. Such methods are informal or semi-formal and rely on a modeller's ability and desire to properly apply the method. In this chapter we discuss the modelling pattern concept enabling formal description of various development methodologies in such a way that the result can be used to assist a modeller in a formal development.

5.2 Modelling Patterns

A software engineering method helps an engineer to decide how to proceed next at any given stage of a system design. The modelling pattern does the same for a formal development. The set of properties and requirements of a system form the initial top-level goal of the development. The goal is matched against known modelling patterns to find a pattern which leads from the abstract to the a model satisfying the system requirements. The modelling pattern mechanism is based on the divide-and-conquer principle. To construct a refinement chain towards some goal we introduce an intermediate goal and try to achieve that goal first. New intermediate goals are introduced until the refinement chain becomes simple enough to be constructed manually or with a combination of refinement patterns.

The role of a modelling pattern is to assist modeller by giving hints on how to construct model refinements. The hints are given in the form of goals stating the desired system properties. A trivial modelling pattern corresponds to an unassisted refinement process where only a global goal is given. A very detailed modelling pattern supported by a collection of refinement patterns automates all or most of a development. A more practical case, however, is an abstract modelling pattern supported by several refinement patterns. Such pattern controls modeller's focus by setting current goals and ensuring that modeller satisfies them while constructing model refinement. An interactive formal development tool can be used to highlight the current goal and the relevant model parts.

A modelling pattern, just like a refinement or abstraction pattern, is concerned with model refinement (abstraction). But unlike refinement pattern, which is an executable procedure computing a new model, modelling patterns is an abstract description of refinement chain. This difference implies different means of descriptions. Modelling pattern is defined by pair of an *assumption* and *goal*. The assumption part is what the kind of models a modelling pattern requires as a starting point. Goal is the kind models the pattern helps to constructs. Goals and assumptions are predicates; they define model classes, just requirements of model transformations and refinement patterns. A simple modelling pattern has the following form

$$\textsf{from } A \textsf{ achieve } G$$

where A is assumption and G is goal. The goal and the assumption predicate take the current model of development as an implicit parameter. Assumption and goal are of the same nature as refinement pattern input and output model classes. However, unlike refinement pattern, a modelling pattern presents no rule for constructing a concrete model. For example, a modelling pattern for the development of a vending machine could have the following form:

$$\textsf{from } true \textsf{ achieve } VM$$

The pattern simply states that goal VM (vending macine construction) is achieved without any assumptions.

Observing that input and output model classes of refinement pattern have the same nature as the assumption and goal of a modelling pattern, it is easy to construct a modelling pattern from a refinement pattern. For refinement pattern p we have modelling pattern from $\mathbf{incl}(p)$ achieve $\mathbf{outcl}(p)$, where $\mathbf{incl}(p)$ and $\mathbf{outcl}(p)$ are the input and output model classes of the pattern. There is, however, no straightforward procedure for producing a refinement or abstraction pattern from a modelling pattern. Moreover, for some modelling patterns there may be no corresponding refinement/abstraction pattern at all[1].

Several trivial modelling patterns can be formed using Boolean constants for goal and assumption. Pattern from $false$ achieve $true$ cannot accept any model but produces all possible models. This pattern serves as a starting point for constructing modelling patterns. Pattern from $false$ achieve $false$ also does not accept any model and its goal describes impossible model. Pattern from $true$ achieve $true$ is a "magic" pattern that is able to construct any model from any model. This pattern is not implementable and any pattern that can be refined to this pattern is also not implementable. The final trivial pattern is from $true$ achieve $false$. This pattern accepts any model but fails to produce anything.

[1]This really means that a modelling has conflicting assumption and goal. In the scope of this work we do not attempt to give a complete formal treatment to this problem.

Complex modelling patterns are constructed using several forms of pattern composition. Sequential composition -

from A_1 achieve G_1 then from A_2 achieve G_2

- describes a modelling pattern which first tries to achieve goal G_1 assuming that a current model satisfies A_1 and then it switches to a new goal G_2 with the assumption condition A_2. Goal G_1 must be stronger than the assumed condition A_2, otherwise the development will stuck after achieving G_1. For our vending machine example, we can use the rule to state that a vendimg machine should be constructed as a modification of a simpler distribution machine:

from $true$ achieve DM then from DM achieve VM

Modelling pattern from $true$ achieve DM address constructing of a distribution machine. Pattern from DM achieve VM adds payment collection functionality converting a distribution machine into a vending machine.

Parallel composition describes two modelling patterns which goals can satisfied independently:

from A_1 achieve G_1 and from A_2 achieve G_2

For legibility, when there more than two parallel rules, we use the following notation:

```
parallel
  from $A_1$ achieve $G_1$
  from $A_2$ achieve $G_2$
```

Such modelling pattern describes a refinement chain leading to model satisfying both G_1 and G_2.

The distribution machine from the vending machine example can be decomposed into a combination of a mechanism for delivering goods to a customer and a loading facility, adding new goods into the machine. Then, the previous modelling pattern for the example can be refiend into

```
parallel
   from $true$ achieve $Delivery$
   from $true$ achieve $Loading$
then
   from $DM$ achieve $VM$
```

provided that $Delivery \wedge Loading \Rightarrow DM$, i.e., the combination of delivery and loading machine is a possible realisation of a distribution machine.

In the case when a modelling pattern describes identical transformation for an array of model elements, it may be convenient to express the pattern as a parallel composition of parametrized patterns:

```
parallel
   from $A(p_1)$ achieve $G(p_1)$  =  parallel forall $p$ where $C(p)$
   ...                                    from $A(p)$ achieve $G(p)$
   from $A(p_k)$ achieve $G(p_k)$
```

where $p_1, \ldots p_k = \{p \mid C(p)\}$.

The choice construct permits construction of a branching pattern that lets a modeller to adapt the pattern to the specifics of a given development. Pattern choice is resolved by matching assumptions of alternatives against the current model:

```
from $A_1$ achieve $G_1$ or from $A_2$ achieve $G_2$
```

The alternative syntax for this construct is

```
choice
   from $A_1$ achieve $G_1$
   from $A_2$ achieve $G_2$
```

The pattern above is equivalent to from A_1 achieve G_1 when only assumption A_1 is satisfied and to from A_2 achieve G_2 when only A_2 is satisfied. When both assumptions are satisfied the choice is resolved by a modeller.

For the vending machine example, we can use the choice construct to define several alternatives for the payment collection service.

```
parallel
   from $true$ achieve $Delivery$
   from $true$ achieve $Loading$
then
   choice
      from $DM$ achieve $CoinsVM$
      from $DM$ achieve $CardVM$
      from $DM$ achieve $FreeVM$
```

where each of $*VM$ goals implies the VM goal.

A simple modelling pattern can be implemented as a refinement pattern. In a place of a modelling pattern from A achieve G we can always use a refinement pattern p, such that $A \Rightarrow \mathbf{incl}(p)$ and $\mathbf{outcl}(p) \Rightarrow G$. In other word, a modelling pattern can be refined into a suitable refinement pattern. This principle is used to automate some of the refinement steps described by a of a modelling pattern so that the pattern is easier to apply in a development.

It is often required to ensure that a certain property is maintained for a number of refinement steps. For this we use a modelling pattern with a property to be maintained as both the assumption and the goal of the pattern:

from P achieve P

This modelling pattern describes a succession of refinement steps preserving property P. For better readability, we use the following syntax for this type of modelling patterns:

maintain P

A typical use for this construct is to require that some property is maintained during a development. For example, the following pattern:

maintain P and from Q achieve R

is applicable to a model satisfying both P and Q and describes a development working towards R while maintianing P. The pattern is equivalent to from $Q \wedge P$ achieve $R \wedge P$ but has the benefit of explicit separation of a property preservation from a development goal.

One possible application of the maintain p construct is to allow several modellers to work on parts of the same development. A development part i is characterised by a property p_i (for Event-B developments, it is convinient to use a facet

(Section 3.5.2) to express a property to be maintained). A modeller working on the $i - th$ part of a development is only allowed to change that part. This condition is formulated as follows

$$\begin{array}{c}\textsf{parallel forall } i \textsf{ where } i \in 1..n \\ \textsf{maintain } \bigwedge_{j \in 1..n \land j \neq i} p_j\end{array}$$

and each modeller k is assigned part maintain $\bigwedge_{j \in 1..n \land j \neq k} p_j$. It may be more useful to include specific goals for each modeller:

$$\begin{array}{l}\textsf{parallel forall } i \textsf{ where } i \in 1..n \\ \quad \textsf{maintain } \bigwedge_{j \in 1..n \land j \neq i} p_j \textsf{ and achieve } G_i\end{array}$$

where G_i is the development goal for the i-th part a development. This is not quite the same as Event-B model decomposition [20, 92], but it is sufficient to support distributed development process, provided a development tool enforces the compliance with modelling pattern goals.

Modelling patterns are constructed independently of the developments in which they are applied and, like refinement patterns, they can be reused in many different developments. This is why we relate them to software engineering methods. Like a method that can be applied to a broad range of problems within some problem domain, modelling patterns are applicable to a class of developments. Several modelling patterns may be combined to produce a complex modelling pattern.

While, in principle, any formal development can be constructed as a sequence of refinement patterns, in practice, it is very unlikely that for any realistic system there will be enough pre-fabricated refinement patterns to construct a development entirely from refinement patterns. Manual refinement steps cannot be avoided. Still, we would like to have many steps of a development automated with refinement patterns. For this we should be able to identify what are the possible refinement steps in a given class of developments and decide which of them can be converted into refinement patterns.

5.2.1 Developing Modelling Patterns

A natural way to construct a modelling pattern is to start with a simple pattern and then detalise it in a gradual manner. The first step of a pattern development is the definition of the global assumptions and goals. Since it is likely that writing detailed assumptions and goals for an abstract pattern is hard, we propose to refine

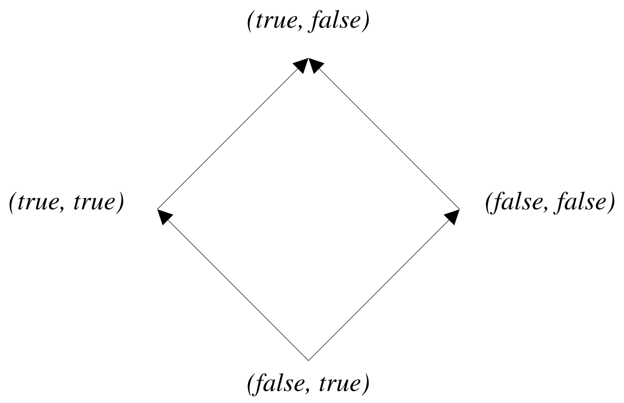


Figure 5.1: Lattice constructed from trivial modelling patterns ordered by the refinement relation.

assumptions and goals along with pattern detalisation by refining the ontology used to express them. Abstract assumptions and goals are specified using In the terms of the problem domain. A pattern is refined until all the phenomena specific to the modelled method are captured. During this process the pattern ontology is extended as needed. Finally, the method-specific taxonomy of the modelled pattern is mapped into terms of the model elements of the chosen formalisation method.

Modelling patterns are constructed using a step-wise refinement procedure. Each step makes pattern more specific. In a general case, an abstract pattern can replaced by another pattern with a weaker assumption or a stronger goal or both:

$$\left(\begin{array}{c} A \Rightarrow A' \\ G' \Rightarrow G \end{array}\right) \Rightarrow \textsf{from}\ A\ \textsf{achieve}\ G \sqsubseteq \textsf{from}\ A'\ \textsf{achieve}\ G'$$

The from $false$ achieve $true$ pattern is refined by any other:

$$\textsf{from}\ false\ \textsf{achieve}\ true \sqsubseteq \textsf{from}\ A\ \textsf{achieve}\ G$$

The from $true$ achieve $false$ is a refinement of any modelling pattern:

$$\textsf{from}\ A\ \textsf{achieve}\ G \sqsubseteq \textsf{from}\ true\ \textsf{achieve}\ false$$

Patterns "from $false$ achieve $true$" and "from $true$ achieve $false$" are the least and the greatest elements of the partial order generated by the modelling pattern refinement (Figure 5.1).

Using the definition of modelling pattern refinement, we introduce several decomposition-based pattern refinement rules. The first such rule decomposes a pattern into a parallel composition of two new patterns. Assumptions of the new patterns must be the same or weaker than the abstract pattern assumption and the conjunction of the new pattern goals must be at least as strong as the abstract goal:

$$\begin{array}{l}\textsf{from } A \textsf{ achieve } G \sqsubseteq \\ \quad \textsf{from } A_1 \textsf{ achieve } G_1 \textsf{ and from } A_2 \textsf{ achieve } G_2, \\ \qquad \text{where} \begin{pmatrix} A \Rightarrow A_1 \\ A \Rightarrow A_2 \\ G_1 \wedge G_2 \Rightarrow G \end{pmatrix}\end{array}$$

Informally, such refinement is interpreted as splitting the abstract pattern into two sub-goals which must be satisfied by the same refinement chain. A more general case is the refinement into the $\begin{array}{l}\textsf{parallel forall } p \textsf{ where } C(p) \\ \quad r\end{array}$ construct

$$\begin{array}{l}\textsf{from } A \textsf{ achieve } G \sqsubseteq \\ \quad \textsf{parallel forall } p \textsf{ where } C(p) \\ \qquad \textsf{from } \mathcal{A}(p) \textsf{ achieve } \mathcal{G}(p) \end{array},$$
$$\text{where} \begin{pmatrix} \forall p \cdot (C(p) \wedge \mathcal{A}(p) \Rightarrow A) \\ \bigwedge_{C(p)} \mathcal{G}(p) \Rightarrow G \end{pmatrix}$$

Sometimes, a modelling pattern can be simplified by applying the opposite of this transformation. Parallel composition of two patterns can be replaced with a single pattern:

$$\begin{array}{l}\textsf{from } A_1 \textsf{ achieve } G_1 \textsf{ and from } A_2 \textsf{ achieve } G_2 \sqsubseteq \\ \quad \textsf{from } A \textsf{ achieve } G, \\ \qquad \text{where} \begin{pmatrix} A_1 \Rightarrow A \\ A_2 \Rightarrow A \\ G \Rightarrow G_1 \wedge G_2 \end{pmatrix}\end{array}$$

A related form of refinement is the decomposition of a pattern into a sequence of patterns. The goal of the last part must be at least as strong as the abstract pattern goal and the assumption of the first pattern in a sequent must be weaker than the abstract assumption. Patterns in a sequent must connect properly, that is, we must ensure that the assumption of the second pattern is satisfied by the goal of the first pattern:

$$\text{from } A \text{ achieve } G \sqsubseteq \\ \quad \text{from } A_1 \text{ achieve } G_1 \text{ then from } A_2 \text{ achieve } G_2, \\ \text{where} \begin{pmatrix} G_2 \Rightarrow G \\ G_1 \Rightarrow A_2 \\ A \Rightarrow A_1 \end{pmatrix}$$

This refinement style introduces milestones that help to arrive to the global goal by adressing a sequence of simpler goals.

An abstract pattern can be decomposed into a choice between alternative patterns

$$\text{from } A \text{ achieve } G \sqsubseteq \\ \quad \text{from } A_1 \text{ achieve } G_1 \text{ or from } A_2 \text{ achieve } G_2, \\ \text{where} \begin{pmatrix} A_1 \vee A_2 \\ A \Rightarrow A_1 \vee A_2 \\ G_1 \Rightarrow G \\ G_2 \Rightarrow G \end{pmatrix}$$

Such refinement makes modelling patterns more flexible. When one tactics cannot be applied, a pattern automatically switches to an alternative tactics. In case when both alternatives are possible, the choice must be resolved externally, e.g. by consulting a modeller.

5.3 Code Generation Case Study

To illustrate the mechanism of modelling patterns we construct a modelling pattern for a simple code generator. First we discuss the method for code generation that we are going to formalise. Then we construct the modelling pattern using the step-wise development procedure we have discussed. Finally, we built a complete design component by adding a number of refinement patterns to automate parts of the modelling pattern. We are able to indetify a subset of the pattern that describes a completely automated code generator.

From formal modelling viewpoint, code generation is a sequence of refinement steps leading to an executable model. A traditionl code generation tool [24] constructs a program in a single step. This makes it difficult to analyse correctness of the produced programs and valide code generators.

In this case study we explore a method construct model implementation in a gradual manner using the refinement technique. We address the following issues

- language-neutral approach - no need to extend a formal method notation with constructs from programming languages;

- support for real-life languages - the target language can be any programming language;
- flexibility - instead of defining a single black box transforming a model into a program, we define many simple transformations that can changed, extended and reused;
- refinement-based - conversion of a model into a program is a sequent of refinement steps.

5.3.1 Event-Basic

We define a programming language called Event-Basic that acts as the target language for our code generation method. Event-Basic [2] is a toy language based on the traditional Basic programming language. It is a very simple language with few constructs

`GOTO` $(label)$	unconditional jump
$(label)$:	label
`INPUT` $(variable)$	console input into a variable
$(variable) = (expression)$	assignment
`IF` $(predicate)$ `THEN` $(statement)$	conditional execution

In the tradition of Basic, Event-Basic is a dynamically typed language. A variable can be used without an explicit prior declaration and the appropriate type is determined automatically. This saves us from dealing with variable declarations required by strongly typed languages.

To avoid interference with programming language structuring constructs (such as the distinction between local and global variables), we use only the simplest control flow statement - unconditional jump. The notable feature of Event-Basic is the assumed support for the complete mathematical language of Event-B. As most imperative programming languages, Event-Basic does not support non-determinism and the only way to update a program variable is to explicitly assign it a new value.

5.3.2 Constructing Model Implementation

Traditional code generator takes a model and produces a program:

[2]Event-Basic is a subset of a real programming language Lemick[93].

$$p : S \nrightarrow P$$

An intermediate modelling language can be used as a bridge between a modelling language and a programming language. For example, language B0 of the B Method adds a number of programming language constructs and additional restrictions to ensure that a model is executable. Construction of an implemenation with such intermediate language is represented as:

$$i \circ p, \text{where } i : S \nrightarrow I \text{ and } p : I \nrightarrow P$$

The approach can generalised so that code generation is done in a number of steps:

$$p = p_1 \circ p_2 \circ \cdots \circ p_n, \text{where} \begin{pmatrix} p_1 : S \nrightarrow A \\ p_2 : A \nrightarrow A \\ \ldots \\ p_n : A \nrightarrow P \end{pmatrix}$$

where A is an abstract implementation and P is the final, executable implementation.

The notion of implementation refinement is based the replacement of abstract parts with more detailed parts, which may also contain abstract elements. This can be related to writing a pseudo-code rendering of a program and then gradually revealing details until all pseudo-code elements are removed.

To simplify reasoning about an Event-Basic program, we convert it into an equivalent Lisp expression (including pseudo-code elements). Then, implementation a is said to be refined by b if for each element of a there is an equal or sub-typed element from b at the same position

$$\mathbf{pref}(a, b) \overset{\text{df}}{=} (car(a) = car(b) \vee car(b)\ \underline{\mathbf{impl}}\ car(a) \wedge \mathbf{pref}(cdr(a), cdr(b))$$

and

$$\mathbf{pref}(a, b) \Rightarrow a \sqsubseteq b$$

Predicate $car(b)\ \underline{\mathbf{impl}}\ car(a)$ states that program part $car(b)$ is a valid implementation of another program part $car(a)$. For example

$$\begin{array}{l} input\_a \\ b = b + a \end{array}$$

may be refined into

$$\begin{array}{l} \texttt{INPUT}\ a \\ b = b + a \end{array}$$

assuming that we know somehow that $\texttt{INPUT}\ a$ $\underline{\textbf{impl}}$ $\texttt{INPUT}\ a$ holds. To simplify the discussion, we do not give formal interpretation to the $\underline{\textbf{impl}}$ relation (this would require use to formally define the language semantics first). Instead, we record the cases of $\underline{\textbf{impl}}$ as assumptions which a modeller has to review, validate and accept prior to applying the developed pattern. Each time a pseudo-code program part is replaced with a concrete one, the corresponding assumption is added as an annotation to the modelling pattern. For the example above, we would add the following annotation

$(\ \underline{\textbf{impl}}\ \ (\texttt{INPUT}\ a)\ input\_a)$

5.3.3 Modelling Pattern

The initial modelling pattern for constructing model implementations has the following definition

$$P_0 = \textsf{from}\ true\ \textsf{achieve}\ \text{defines}(\textsf{Impl})$$

which reads as *"take an arbitrary model and construct and an Event-Basic implementation"* where the phrase *"Event-Basic implementation"* is our informal interpretation of goal EventBasicImpl. To formulate this goal precisely we investigate what does it means to construct an Event-Basic implementation for an Event-B model. For brevity, defines is omitted in goals and assumptions:

$$P_0 = \textsf{from}\ true\ \textsf{achieve}\ \textsf{Impl}$$

An Event-B system is an endless loop executing model events [20, pages 3,4]. System operates as long as there is an enabled event and stops when no event can be executed due to guard restrictions. An event to be executed is randomly selected among the enabled events of a system. Further, event parameters are selected randomly from the set of possible parameter values, as defined by an event guard. In

Event-Basic the global system loop is implemented using the `GOTO` statement returning control to the beginning of a program at the end of each execution cycle. When no event is enabled, a program constructed with this method deadlocks by endlessly executing the `GOTO` statement [3]:

$$\mathsf{GotoImpl} = \begin{bmatrix} \mathsf{Init} \\ :\texttt{.start} \\ \mathsf{Events} \\ \texttt{GOTO}\ \texttt{.start} \end{bmatrix}$$

Here Events is a program block implementing all the system events and Init is a block initialising program variables. The above formulates one of the possible top-level architectures for Event-Basic implementations. An important fact about this particular architecture is that we make an assumption that an endless execution of jump is a suitable implementation of deadlock. This architecture is a detalisation of the $Impl$ goal:

$$\mathsf{GotoImpl}\ \underline{\mathbf{impl}}\ \mathsf{Impl}$$

GotoImpl is a stronger goal, hence the initial pattern definition can changed to

$$P_1 = \textsf{from}\ true\ \textsf{achieve}\ \mathsf{GotoImpl}$$
$$P_0 \sqsubseteq P_1$$

The definition of GotoImpl refers to the initialisation block Init and event block Events. In other words, for the GotoImpl goal to be reached, the Init and Events goals must be reached first. These two goals can be satisfied independentl using the parallel composition. The new modelling patten is

$$P_2 = \begin{array}{l} \textsf{parallel} \\ \quad \textsf{from}\ true\ \textsf{achieve}\ \mathsf{Init} \\ \quad \textsf{from}\ true\ \textsf{achieve}\ \mathsf{Events} \\ \textsf{then} \\ \quad \textsf{from}\ \mathsf{Init} \wedge \mathsf{Events}\ \textsf{achieve}\ \mathsf{GotoImpl} \end{array}$$
$$P_1 \sqsubseteq P_2$$

[3] Strictly speaking, this is another assumption that should be added to the modelling pattern as annotation.

To keep modelling pattern expressions legible, we focus on important sub-patterns and refine them independently. These sub-patterns are the initialisation pattern $I_0 = (true, \mathsf{Init})$ and the event block $E_0 = (true, \mathsf{Events})$:

$$\begin{array}{l} P_2 = I_0 \\ E_0 \ \textsf{and} \ \ \textsf{then from} \ \mathsf{Init} \wedge \mathsf{Events} \ \textsf{achieve} \ \mathsf{GotoImpl} \end{array}$$

Also, by $E_i, i > 0$ we mean modelling pattern refining E_{i-1} and, by transitivity, E_0. The same applies to notation I_i.

Two major parts of an implementation are the event selection block and event body implementations. The event selection block inputs event parameters, evaluates event guards and passes control to the currently enabled event. Event body code updates program variables corresponding to the variables of the model:

$$\mathsf{EventsImpl} = \begin{bmatrix} \mathsf{EventSelection} \\ \texttt{GOTO start} \\ \mathsf{EventBodies} \end{bmatrix}, \text{where } \mathsf{EventsImpl} \ \underline{\mathbf{impl}} \ \mathsf{Events}$$

and

$$E_1 = \begin{array}{l} \textsf{parallel} \\ \quad \textsf{from} \ true \ \textsf{achieve} \ \mathsf{EventSelection} \\ \quad \textsf{from} \ true \ \textsf{achieve} \ \mathsf{EventBodies} \\ \textsf{then} \\ \quad \textsf{from} \ \mathsf{EventSelection} \wedge \mathsf{EventBodies} \ \textsf{achieve} \ \mathsf{EventsImpl} \end{array}$$

The event selection block passes control to an enabled event. This is implemented as a sequential composition of selectors for all the individual system events:

$$\mathsf{EventSelectionImpl} = \begin{bmatrix} \mathsf{EventSel}(e_1) \\ \ldots \\ \mathsf{EventSel}(e_n) \end{bmatrix} \wedge \mathsf{EventSelectionImpl} \ \underline{\mathbf{impl}} \ \mathsf{EventSelection}$$

The new modelling pattern is

$$E_2 = \begin{array}{l} \textsf{parallel} \\ \quad \textsf{from} \ true \ \textsf{achieve} \ \mathsf{EventSelectionImpl} \\ \quad \textsf{from} \ true \ \textsf{achieve} \ \mathsf{EventBodies} \\ \textsf{then} \\ \quad \textsf{from} \ \mathsf{EventSelection} \wedge \mathsf{EventBodies} \ \textsf{achieve} \ \mathsf{EventsImpl} \end{array}$$

We know that the EventSelectionImpl goal can reached by producing selection block for all the model events, except the initialisation event. This is expressed with the following new pattern:

$$E_3 = \begin{array}{l} \textsf{parallel} \\ \quad \textsf{parallel forall } e \textsf{ where } e \in evt_n \textsf{ do from } true \textsf{ achieve } \mathsf{EventSel}(e) \\ \quad \textsf{from } true \textsf{ achieve } \mathsf{EventBodies} \\ \textsf{then} \\ \quad \textsf{from } \mathsf{EventSelection} \wedge \mathsf{EventBodies} \textsf{ achieve } \mathsf{EventsImpl} \end{array}$$

where set $evt_n = evt \setminus \{ini\}$ is the set of all model events excluding the initialisation event. The block of event bodies is made of implementations of individual event bodies:

$$\mathsf{EventBodiesImpl} = \begin{bmatrix} \mathsf{EventBody}(e_1) \\ \dots \\ \mathsf{EventBody}(e_n) \end{bmatrix} \wedge \mathsf{EventBodiesImpl} \ \underline{\textbf{impl}} \ \mathsf{EventBodies}$$

The goal EventBodies is reached by implementing the body of each event:

$$E_4 = \begin{array}{l} \textsf{parallel} \\ \quad \textsf{parallel forall } e \textsf{ where } e \in evt_n \textsf{ do from } true \textsf{ achieve } \mathsf{EventSel}(e) \\ \quad \textsf{parallel forall } e \textsf{ where } e \in evt_n \textsf{ do from } true \textsf{ achieve } \mathsf{EventBody}(e) \\ \quad \textsf{then} \\ \quad\quad \textsf{from } \mathsf{EventBodiesImpl} \textsf{ achieve } \mathsf{EventBodies} \\ \textsf{then} \\ \quad \textsf{from } \mathsf{EventSelection} \wedge \mathsf{EventBodies} \textsf{ achieve } \mathsf{EventsImpl} \end{array}$$

Further we focus on the event body and event selector parts. Let

$$\begin{array}{lcl} Sel_0 & = & \textsf{parallel forall } e \textsf{ where } evt_n \textsf{ do from } true \textsf{ achieve } \mathsf{EventSel}(e) \\ Bod_0 & = & \textsf{parallel forall } e \textsf{ where } evt_n \textsf{ do from } true \textsf{ achieve } \mathsf{EventBody}(e) \end{array}$$

then the modelling pattern above becomes

$$E_4 = \begin{array}{l} \textsf{parallel} \\ \quad Sel_0 \\ \quad Bod_0 \textsf{ then from } \mathsf{EventBodiesImpl} \textsf{ achieve } \mathsf{EventBodies} \\ \textsf{then} \\ \quad \textsf{from } \mathsf{EventSelection} \wedge \mathsf{EventBodies} \textsf{ achieve } \mathsf{EventsImpl} \end{array}$$

To implement event selector we first some values for the event parameters and then evaluate the event guard to determine whether to execute event or not. A more detailed version of an event selector is

$$\mathsf{EvtSelImpl}(e) = \left[\begin{array}{l} ParInput(e) \\ BlockSelection(e) \end{array}\right] \wedge \mathsf{EvtSelImpl}\ \ \underline{\textbf{impl}}\ \ \mathsf{EvtSel}$$

The EventSelImpl goal is reached by first addressing goals ParamInput and then BlockSelection. These goals, in their turn, must be reachable for any model:

$$\textsf{from}\ true\ \textsf{achieve}\ \mathsf{EventSel}(e) \sqsubseteq SL_0(e)$$

where $SL_0(e)$ is defined as

```
            parallel
              from $true$ achieve ParInput($e$)
$SL_0(e) =$     from $true$ achieve BlockSelection($e$)
            then
              from ParInput($e$) $\wedge$ BlockSelection($e$) achieve EventSelImpl($e$)
```

Sub-pattern Sel_0 is now refined into

$$Sel_1 = \textsf{parallel forall}\ e\ \textsf{where}\ evt_n\ \textsf{do from}\ true\ \textsf{achieve}\ SL_0(e)$$

For parameter initialisation we rely on the `INPUT` statement. The statement reads values into variables from some external information source such as a file, keyboard or another program. The statement is unable to control the range of inputted values so we rely on the event guard to filter out inappropriate parameters. We hypothesize that `INPUT` is *fair*. That is, any given set of event parameters is generated after sufficiently many attempts. In addition, we require that event parameters are defined on implementable domains.

$$\mathsf{ParInputImpl}(e) = \left[\begin{array}{c} \texttt{INPUT}\ p^e_1 \\ \ldots \\ \texttt{INPUT}\ p^e_m \end{array}\right] \wedge \mathsf{ParInputImpl}\ \ \underline{\textbf{impl}}\ \ \mathsf{ParInput}$$

where $\{p^e_1, ..., p^e_m\} = e.\mathrm{arg}$. Implementable event parameter is characterised by the following predicate

$$ImDom(u) \overset{\text{df}}{=} \exists m, t \cdot (u \in t \wedge m : t \rightarrowtail 0..cmax)$$

where $cmax > 0$ is the maximum natural implementable value for a given target implementation platform (e.g. 2^{32}). An event can be transformed by the pattern if all the event parameters are implementable

$$GdPar(e) \overset{\text{df}}{=} \forall p \cdot (p \in e.\text{arg} \Rightarrow ImDom(p))$$

And the new event selection sub-pattern is

```
            parallel
               parallel
                  from GdPar achieve ParInputImpl($e$)
$SL_1(e) =$       from ¬GdPar achieve ParInput($e$)
               from $true$ achieve BlockSelection($e$)
            then
               from ParInput($e$) ∧ BlockSelection($e$) achieve EventSelImpl($e$)
```

Note that though we are unable not address the case of non-implementable parameters in Event-Basic we still have to include a modeling pattern to cover this case. Pattern $(\neg\mathsf{GdPar}, \mathsf{ParInput}(e))$ transforms non-implementable parameters into a parameter initialisation block.

Control to an enabled event is passed using the `IF` statement. The statement evaluates the guard of a corresponding event and, if the guard is true, jumps to the block implementing the event body. An event guard, being an expression, is always computable.

Body of an even is made of assignments to the variables updated in the event. The block of assignments is preceded with a label that is used by the corresponding `IF` statement.

$$\mathsf{BlockSelectionImpl}(e) = \left[\ \texttt{IF}\ e.\text{guard}\ \texttt{THEN GOTO}\ e.\text{nom}\ \right] \wedge$$
$$\mathsf{BlockSelectionImpl}\ \underline{\textbf{impl}}\ \mathsf{BlockSelection}$$

$$\mathsf{EventBodyImpl}(e) = \begin{bmatrix} :e.\text{nom} \\ \mathsf{Assignment}(a_1) \\ \dots \\ \mathsf{Assignment}(a_k) \end{bmatrix} \wedge \mathsf{EventBodyImpl}\ \underline{\textbf{impl}}\ \mathsf{EventBody}$$

Here $\{a_1, ...a_k\} = e.\text{act}$. Note that we have to assume that no model event can have name *.start* as this name is reserved for the label pointing at the program beginning. This is true at least for the current version of Event-B. In the event selection sub-pattern the old BlockSelection goal is replaced with the BlockSelectionImpl goal

$$SL_2(e) = \begin{array}{l} \textsf{parallel} \\ \quad \textsf{parallel} \\ \quad\quad \textsf{from GdPar achieve ParInputImpl}(e) \\ \quad\quad \textsf{from } \neg\textsf{GdPar achieve ParInput}(e) \\ \quad \textsf{from } \mathit{true} \textsf{ achieve BlockSelectionImpl}(e) \\ \textsf{then} \\ \quad \textsf{from ParInput}(e) \wedge \textsf{BlockSelection}(e) \textsf{ achieve EventSelImpl}(e) \end{array}$$

Pattern $(\mathit{true}, \textsf{EventBody}(e))$ is refined to show that an implementation of an event body is made of individual assignments to variables

$$\begin{array}{l} \textsf{from } \mathit{true} \textsf{ achieve EventBody}(e) \sqsubseteq \\ \quad \textsf{from } \mathit{true} \textsf{ achieve EventBodyImpl}(e) \sqsubseteq \\ \quad\quad \textsf{parallel forall } a \textsf{ where } a \in a.\mathrm{act} \textsf{ do from } \mathit{true} \textsf{ achieve Assignment}(a) \end{array}$$

Finally, a variable assignment is constructed for implementable event actions

$$\textsf{AssignmentImpl}(a) = \left[\ \mathit{var}(a) = a.\mathrm{exp}\ \right] \wedge \textsf{AssignmentImpl}\ \ \underline{\textbf{impl}}\ \ \textsf{Assignment}$$

Here $\mathit{var}(a)$ is defined by the relation $\{\mathit{var}(a)\} = a.\mathrm{var}$. An action is implementable if it is a deterministic substitution assigning to a single variable (for simplicity we choose not to deal with multiple variable substitution)

$$\mathit{GdSubst}(a) \ \overset{\mathrm{df}}{=}\ a.\mathrm{sty} = (:=) \wedge a.\mathrm{sty} = \{v\} \wedge \mathit{ImDom}(v)$$

Pattern $(\mathit{true}, \textsf{Assignment}(e))$ is refined as follows

$$\textsf{from } \mathit{true} \textsf{ achieve Assignment}(a) \sqsubseteq \begin{array}{l} \textsf{parallel} \\ \quad \textsf{from GdSubst}(a) \textsf{ achieve AssignmentImpl}(a) \\ \quad \textsf{from } \neg\textsf{GdSubst}(a) \textsf{ achieve Assignment}(a) \end{array}$$

where pattern $(\neg\textsf{GdSubst}(a), \textsf{Assignment}(a))$ caters for non-deterministic actions and multiple variable actions.

The initialisation event is a special model event which has no guard and which sole purpose is to compute initial values for all the system variables. The body of such event is no different from the body of a normal event, hence we can reuse the modelling patterns constructed for normal model events. Let *ini* be the name of the initialisation event. Then

```
$I_1$ = from $true$ achieve EventBody($ini$) then
          from EventBody($ini$) achieve Init
--------------------------------------------------------------------------------
$I_2$ = from $true$ achieve EventBodyImpl($ini$) then
          from EventBody($ini$) achieve Init
--------------------------------------------------------------------------------
$I_3$ = parallel forall $a$ where $a \in ini.\mathrm{act}$ do from $true$ achieve Assignment($a$) then
          from EventBody($ini$) achieve Init
--------------------------------------------------------------------------------
$I_4$ = parallel forall $a$ where $a \in ini.\mathrm{act}$
          parallel
            from GdSubst($a$) achieve AssignmentImpl($a$)
            from $\neg$GdSubst($a$) achieve Assignment($a$)
        then
          from EventBody($ini$ achieve Init
```

The complete modelling pattern is the result of the refinement of different sub-patterns

```
$P_f =$
parallel
   parallel forall $e$ where $e \in evt_n$
      parallel
         from GdPar$(e)$ achieve ParInputImpl$(e)$
         from $\neg$GdPar$(e)$ achieve ParInput$(e)$
         from $true$ achieve BlockSelectionImpl$(e)$
      then
         from ParInput$(e) \wedge$ BlockSelection$(e)$ achieve EventSelImpl$(e)$
   parallel forall $e, a$ where $e \in evt_n \wedge a \in e.\mathrm{act}$
      parallel
         from GdSubst$(a)$ achieve AssignmentImpl$(a)$
         from $\neg$GdSubst$(a)$ achieve Assignment$(a)$
   then
      from EventBodiesImpl achieve EventBodies
   parallel forall $a$ where $a \in ini.\mathrm{act}$
      parallel
         from GdSubst$(a)$ achieve AssignmentImpl$(a)$
         from $\neg$GdSubst$(a)$ achieve Assignment$(a)$
   then
      from EventBody$(ini)$ achieve Init
then
   from EventSelection $\wedge$ EventBodies achieve EventsImpl
then
   from Init $\wedge$ Events achieve GotoImpl
```

The pattern can be simplified by noticing that

```
(parallel
   parallel forall $e, a$ where $e \in evt_n \wedge a \in e.\mathrm{act}$
      parallel
         from GdSubst($a$) achieve AssignmentImpl($a$)
         from $\neg$GdSubst($a$) achieve Assignment($a$)
   then
      from EventBodiesImpl achieve EventBodies
   parallel forall $a$ where $a \in ini.\mathrm{act}$
      parallel
         from GdSubst($a$) achieve AssignmentImpl($a$)
         from $\neg$GdSubst($a$) achieve Assignment($a$)
   then
      from EventBody($ini$) achieve Init                    )
```

$$=$$

```
(parallel forall $e, a$ where $e \in evt \wedge a \in e.\mathrm{act}$
    parallel
       from GdSubst($a$) achieve AssignmentImpl($a$)
       from $\neg$GdSubst($a$) achieve Assignment($a$)
 then
    from EventBodiesImpl achieve EventBodies           )
```

And the overall pattern is

```
$P_f$ =
parallel
   parallel forall $e$ where $e \in evt_n$
      parallel
         from GdPar($e$) achieve ParInputImpl($e$)
         from $\neg$GdPar($e$) achieve ParInput($e$)
         from $true$ achieve BlockSelectionImpl($e$)
      then
         from ParInput($e$) $\wedge$ BlockSelection($e$) achieve EventSelImpl($e$)
   parallel forall $e, a$ where $e \in evt \wedge a \in e.\mathrm{act}$
      parallel
         from GdSubst($a$) achieve AssignmentImpl($a$)
         from $\neg$GdSubst($a$) achieve Assignment($a$)
   then
      from EventBodiesImpl achieve EventBodies
then
   from EventSelection $\wedge$ EventBodies achieve EventsImpl
then
   from Init $\wedge$ Events achieve GotoImpl
```

The pattern relies on the following set of assumptions

$\mathsf{GotoImpl}\ \underline{\textbf{impl}}\ \mathsf{Impl}$
$\mathsf{EventsImpl}\ \underline{\textbf{impl}}\ \mathsf{Events}$
$\mathsf{EventSelectionImpl}\ \underline{\textbf{impl}}\ \mathsf{EventSelection}$
$\mathsf{EventBodiesImpl}\ \underline{\textbf{impl}}\ \mathsf{EventBodies}$
$\mathsf{EvtSelImpl}\ \underline{\textbf{impl}}\ \mathsf{EvtSel}$
$\mathsf{ParInputImpl}\ \underline{\textbf{impl}}\ \mathsf{ParInput}$
$\mathsf{BlockSelectionImpl}\ \underline{\textbf{impl}}\ \mathsf{BlockSelection}$
$\mathsf{EventBodyImpl}\ \underline{\textbf{impl}}\ \mathsf{EventBody}$
$\mathsf{AssignmentImpl}\ \underline{\textbf{impl}}\ \mathsf{Assignment}$

The assumptions express our informal notion of pseudo-code and program refinement. Further we automate the refinement steps described by the pattern with a set of refinement patterns.

5.3.4 Refinement Patterns

The first pattern decorates all implementable actions with equivalent variable assignments. It implements the following modelling pattern

```
parallel forall $e, a$ where $e \in evt \land a \in e.\mathrm{act}$
  from $\mathsf{GdSubst}(a)$ achieve $\mathsf{AssignmentImpl}(a)$
```

```
pattern $DoAssignment$

  forall $e, a, va$ where
      $e \in evt \land a.\mathrm{var} = \{va\} \land \mathsf{GdSubst}(a)$
  do
      $annotate(a, \texttt{assignment}, [\![va = a.\mathrm{exp}]\!])$
  end
```

To prove that the refinement pattern properly implements the original modelling pattern, we have to demonstrate that the modelling pattern assumption implies the refinement pattern requirements and models constructed by the refinement pattern satisfy the modelling pattern goal. For the former the proof obligation is

$$\begin{array}{l}\mathsf{GdSubst}(a) \Rightarrow in(DoAssignment)\\ \quad \vdash \mathsf{GdSubst}(a) \Rightarrow a.\mathrm{var} = \{va\} \land \mathsf{GdSubst}(a)\end{array}$$

which is demonstrated by expanding GdSubst and noticing that v and va are the same variable. The second proof obligation is

$out(DoAssignment) \Rightarrow \forall e, a \cdot (e \in evt \wedge a \in e.\mathrm{act} \Rightarrow \mathsf{AssignmentImpl}(a))$
$\quad \vdash \forall e, a, va \cdot (a.\mathrm{var} = \{va\} \wedge \mathsf{GdSubst}(a) \Rightarrow \mathrm{defines}(a, [\![va = a.\mathrm{exp}]\!])) \Rightarrow$
$\quad\quad \forall e, a \cdot (e \in evt \wedge a \in e.\mathrm{act} \Rightarrow \mathsf{AssignmentImpl}(a))$
$\quad\quad\quad \vdash \mathbf{pref}(\mathsf{AssignmentImpl}(a), \mathrm{defines}(a, [\![va = a.\mathrm{exp}]\!]))$
$\quad\quad\quad\quad \vdash \mathbf{pref}([\![var(a) = a.\mathrm{exp}]\!], [\![va = a.\mathrm{exp}]\!])$
$\quad\quad\quad\quad\quad \vdash \mathbf{pref}(`(= var(a)\ a.\mathrm{exp}), `(=\ va\ a.\mathrm{exp}))$
$\quad\quad\quad\quad\quad\quad \vdash \{var(a)\} = a.\mathrm{var} \wedge a.\mathrm{var} = \{va\} \wedge (var(a) = va) \wedge (a.\mathrm{exp} = a.\mathrm{exp})$
$\quad\quad\quad\quad\quad\quad \vdash (va = va) \wedge (a.\mathrm{exp} = a.\mathrm{exp})$

Pattern from $\mathsf{GdPar}(e)$ achieve $\mathsf{ParInputImpl}(e)$ automates the parameter input refinement step.

```
pattern $DoParInput$
    with $e, p$ where
        $e \in evt_n \wedge p \in e.\mathrm{arg} \wedge \mathsf{GdPar}(e)$
    do
        $annotate(e, \texttt{input}, [\![\texttt{INPUT}\ p]\!])$
    end
```

Pattern from $true$ achieve $\mathsf{BlockSelectionImpl}(e)$ constructs the refinement step for the selection of a block implementing an event body.

```
pattern $DoBlockSelection$
    with $e$ where
        $e \in evt_n$
    do
        $annotate(e, \texttt{dispatcher}, [\![\texttt{IF}\ e.\mathrm{grd}\ \texttt{THEN GOTO}\ e.\mathrm{nam}]\!])$
    end
```

The two refinement patterns above are combined to implement the following modelling pattern:

```
parallel forall $e$ where $e \in evt_n$
    parallel
        from $\mathsf{GdPar}(e)$ achieve $\mathsf{ParInputImpl}(e)$
        from $true$ achieve $\mathsf{BlockSelectionImpl}(e)$
```

The resulting refinement pattern is

```
pattern $Dispatcher$
  forall $e, p$ where
    $e \in evt_n \wedge p \in e.\mathrm{arg} \wedge \mathsf{GdPar}(e)$
  do
    $annotate(e, \mathtt{input}, [\![\mathtt{INPUT}\ p]\!])$
  end
  forall $e$ where
    $e \in evt_n$
  do
    $annotate(e, \mathtt{dispatcher}, [\![\mathtt{IF}\ e.\mathrm{grd}\ \mathtt{THEN\ GOTO}\ e.\mathrm{nam}]\!])$
  end
```

Pattern

$$\text{from } \mathsf{ParInput}(e) \wedge \mathsf{BlockSelection}(e) \text{ achieve } \mathsf{EventSelImpl}(e)$$

glues together parameter input and *if* statement code generated for all the model events. The pattern constructs a new event and annotates it with event selection code.

```
pattern $DoEventSel$
  with $n$ where
    $n \in \mathrm{Event} \wedge$
  do
    $newevent(n)$
      forall $e$ where
        $e \in evt_n \wedge defines(e, \mathtt{dispatcher})$
      do
        $deannotate(e, \mathtt{dispatcher})$
        $annotate(n, \mathtt{eventsselector}, \left[\!\!\left[ \begin{array}{l} e.\mathtt{input} \\ e.\mathtt{dispatcher} \end{array} \right]\!\!\right])$
      end
  end
```

Pattern from $\mathsf{EventBodiesImpl}$ achieve $\mathsf{EventBodies}$ assembles assignments implementing individual event actions into blocks of assignments, one per each event.

```
pattern $DoEvents$
  forall $e, a$ where
    $e \in evt \wedge a \in ini.\mathrm{action} \wedge defines(a, \mathtt{assignment})$
  do
    $deannotate(a, \mathtt{assignment})$
    $annotate(e, \mathtt{assignblock}, [\![e.\mathtt{assignment}]\!])$
  end
```

Pattern from EventSelection $\wedge$ EventBodies achieve EventsImpl combines implementation of event selection and event bodies.

```
pattern $Assemble$
  forall $e$ where
     $e \in evt\_n \wedge defines(e, \texttt{eventsselector})$
  do
     $deannotate(e, \texttt{eventsselector})$
     $annotate(e, \texttt{programbody}, [\![\ e.\texttt{eventsselector}\ ]\!])$
       $annotate(e, \texttt{programbody}, [\![\texttt{GOTO .start}]\!])$
             forall $ev$ where
                $ev \in Event \wedge defines(ev, \texttt{assignblock})$
             do
                $deannotate(ev, \texttt{assignblock})$
                $annotate\left(e, \texttt{programbody}, \left[\!\!\left[\begin{array}{l} :e.\text{nam} \\ e.\texttt{assignblock} \\ \texttt{GOTO .start} \end{array}\right]\!\!\right]\right)$
             end
  end
```

Pattern from Init $\wedge$ Events achieve GotoImpl produces the final program by putting together initialisation and model implementation.

```
pattern $Link$
  with $e$ where
     $e \in evt_n \wedge defines(e, \texttt{programbody})$
  do
     $deannotate(e, \texttt{programbody})$
     $deannotate(ini, \texttt{assignblock})$
     $annotate\left(e, \texttt{program}, \left[\!\!\left[\begin{array}{l} ini.\texttt{assignblock} \\ :\texttt{.start} \\ e.\texttt{programbody} \end{array}\right]\!\!\right]\right)$
  end
```

5.3.5 Automated Code Generator

Not all the steps of modelling pattern P_f are implemented with the refinement patterns. There are two steps that cannot be easily translated into refinement patters: the initialisation of parameters defined on non-implementable domains and the translation of non-implementable actions into assignments. These cases must be handled with manual refinement steps.

We can also choose to ignore such cases and only provide code generation for a subset of input models accepted by P_i. For this we split the modelling pattern into two cases - one dealing with models that can be translated automatically and the other addressing models that require manual refinement steps. Pattern P_f is refined as follows

```
           choice
P_c =        from ¬GdModel achieve ¬GdModel then $P_f$
             from GdModel achieve GdModel then $P_f$
```

where assumption GdModel describes a model with implementable actions and initialisable parameters

$$GdModel \overset{\text{df}}{=} \quad \forall e \cdot (e \in evt \Rightarrow \forall a \cdot (a \in e.\text{act} \Rightarrow GdSubst(a)) \wedge GdPar(e))$$

Writing out pattern $((\neg\text{GdModel}, \neg\text{GdModel}); P_f)$, we get the following pattern

```
        parallel
          parallel forall $e$ where $e \in evt_n$
            parallel
              from GdPar$(e)$ achieve ParInputImpl$(e)$
              from $true$ achieve BlockSelectionImpl$(e)$
            then
              from ParInput$(e) \wedge$ BlockSelection$(e)$ achieve EventSelImpl$(e)$
$P_i =$   parallel forall $e, a$ where $e \in evt \wedge a \in e.\text{act}$
            from GdSubst$(a)$ achieve AssignmentImpl$(a)$
          then
            from EventBodiesImpl achieve EventBodies
        then
          from EventSelection $\wedge$ EventBodies achieve EventsImpl
        then
          from Init $\wedge$ Events achieve GotoImpl
```

Unlike the more general P_f pattern, P_i is completely automatic.

```
        parallel
          $Dispatcher$ then $DoEventSel$
          parallel forall $e, a$ where $e \in evt \wedge a \in e.\text{act}$ do $DoAssignment(a)$
          then
$P_i =$     $DoEvents$
        then
          $Assemble$
        then
          $Link$
```

5.3.6 Sample Development

In this section we take a simple Event-B model and refine it with the P_i modelling pattern and the related refinement patterns. The result is a sucession of refinement steps leading to a model annotated with an Event-Basic program.

The initial model is the Euclid's algorithm for computing the greatest common divisor of two natural numbers.

```
SYSTEM gcd
VARIABLES $a, b$
INVARIANT $a \in \mathbb{N} \wedge b \in \mathbb{N}$
INITIALISATION $a := 0 \| b := 0$
EVENTS
    input   =  ANY $f, s$ WHERE
                 $a + b = 0 \wedge f + s > 0$
               THEN
                 $a := f$
                 $b := s$
               END
    eucgcd  =  WHEN
                 $b \neq 0$
               THEN
                 $b := a \bmod b$
                 $a := b$
               END
```

The first refinement annotates event actions with the equivalent assignments in the Event-Basic syntax. It is produced by applying pattern *DoAssignment*.

```
SYSTEM $gcd1$
REFINES $gcd$
$\ldots$
INITIALISATION
    $a := 0$
        ANNOT $assignment$
            $a = 0$
        END
    $b := 0$
        ANNOT $assignment$
            $b = 0$
        END
EVENTS
    $input$    $=$  ANY $f, s$ WHERE
                      $a + b = 0 \wedge f + s > 0$
                    THEN
                      $a := f$
                          ANNOT $assignment$
                              $a = f$
                          END
                      $b := s$
                          ANNOT $assignment$
                              $b = s$
                          END
                    END
    $eucgcd$   $=$  WHEN
                      $b \neq 0$
                    THEN
                      $b := a \bmod b$
                          ANNOT $assignment$
                              $b = a \bmod b$
                          END
                      $a := b$
                          ANNOT $assignment$
                              $a = b$
                          END
                    END
```

The second refinement introduces code which chooses between program parts implementing different model events. This refinement step is the result of the *Dispatcher* pattern.

```
SYSTEM $gcd2$
REFINES $gcd1$
. . .
EVENTS
    $input$   =  ANY $f, s$ WHERE $a + b = 0 \wedge f + s > 0$ THEN . . . END
                    ANNOT $dispatcher$
                        IF $a + b = 0 \wedge f + s > 0$ THEN GOTO input
                    END
                    ANNOT $input$
                        INPUT $f$
                        INPUT $s$
                    END
    $eucgcd$  =  WHEN $b \neq 0$ THEN . . . END
                    ANNOT $dispatcher$
                        IF $b \neq 0$ THEN GOTO eucgcd
                    END
```

The third refinement is constructed with pattern *DoEventSel* by putting together the event selection code for all the model events into a single block of code.

```
SYSTEM $gcd3$
REFINES $gcd2$
. . .
EVENTS
    $impl$    =  SKIP
                    ANNOT $eventsselector$
                        INPUT $f$
                        INPUT $s$
                        IF $a + b = 0 \wedge f + s > 0$ THEN GOTO input
                        IF $b \neq 0$ THEN GOTO eucgcd
                    END
    $input$   =  ANY $f, s$ WHERE $a + b = 0 \wedge f + s > 0$ THEN . . . END
    $eucgcd$  =  WHEN $b \neq 0$ THEN . . . END
```

The fourth refinement combines assignments into code blocks corresponding to event bodies. This step is produced with the *DoEvents* refinement pattern.

```
SYSTEM $gcd4$
REFINES $gcd3$
...
INITIALISATION
    $a := 0$
    $b := 0$
    ANNOT $assignblock$
        $a = 0$
        $b = 0$
    END
EVENTS
    $impl$     =  SKIP
                      ANNOT $eventsselector$
                          ...
                      END
    $input$    =  ANY $f, s$ WHERE
                    $a + b = 0 \land f + s > 0$
                  THEN
                    $a := f$
                    $b := s$
                  END
                      ANNOT $assignblock$
                          $a = f$
                          $b = s$
                      END
    $eucgcd$   =  WHEN
                    $b \neq 0$
                  THEN
                    $b := a \bmod b$
                    $a := b$
                  END
                      ANNOT $assignblock$
                          $b = a \bmod b$
                          $a = b$
                      END
```

The fifths refinement puts different program parts together and removes the intermediate annotations. The initialisation event is not affected.

```
SYSTEM $gcd5$
REFINES $gcd4$
...
INITIALISATION
    $a := 0$
    $b := 0$
    ANNOT $assignblock$
        $a = 0$
        $b = 0$
    END
EVENTS
    $impl$      $=$   SKIP
                          ANNOT $programbody$
                              INPUT $f$
                              INPUT $s$
                              IF $a + b = 0 \wedge f + s > 0$ THEN GOTO input
                              IF $b \neq 0$ THEN GOTO eucgcd
                              GOTO .start: input
                              $a = f$
                              $tb = s$
                              GOTO .start
                              : eucgcd
                              $b = a \bmod b$
                              $a = b$
                              GOTO .start
                          END
    $input$     $=$   ANY $f, s$ WHERE $a + b = 0 \wedge f + s > 0$ THEN ... END
    $eucgcd$    $=$   WHEN $b \neq 0$ THEN ... END
```

The last refinement step constructs the complete program by adding the initialisation part.

```
SYSTEM $gcd6$
REFINES $gcd5$
...
EVENTS
  $impl$  =  SKIP
               ANNOT $program$
                 $a = 0$
                 $b = 0$
                 : .start
                 INPUT $f$
                 INPUT $s$
                 IF $a + b = 0 \wedge f + s > 0$ THEN GOTO input
                 IF $b \neq 0$ THEN GOTO eucgcd
                 GOTO .start
                 : input
                 $a = f$
                 $b = s$
                 GOTO start
                 : eucgcd
                 $b = a \bmod b$
                 $a = b$
                 GOTO .start
               END
  ...
```

5.4 Summary

Just like a complex software system is constructed with a help of software engineered methods, a complex development should be constructed using a set of domain-specific modelling tactics. Such tactics, called a modelling pattern, guides a modeller by setting context-specific development goals. A goal is simply a model class or a predicate describing model properties. An essential property of a modelling pattern is that it is machine intepretable. We believe there is a substantial benefit in having a tool to impose a modelling tactics over expecting a modeller to follow informal guidelines.

In this chapter we have discussed a way to define modelling patterns using a set of decomposition rules. The approach we presented was inspired by the goal-driven requirements engineering [69, 94]. However, while requirements engineering is concerned with the construction of a system, development of a modelling pattern leads to a reusable modelling tactics.

We have demonstrated the applicability of the modelling pattern approach by developing a fairly detailed modelling pattern describing a step-wise code generation method. The method advices a modeller on how to convert an Event-B model into a program in the Event-Basic programming language.

Chapter 6

Evaluation

6.1 Introduction

In this chapter we discuss several examples illustrating our approach. The first example is the N-versioning refinement pattern which helps to correctly apply the N-versioning mechanism in formal developments. The pattern is able to automatically construct a refinement working with an arbitrary number of versions.

We continue with the discussion of parity check bit and hamming coding patterns. These two patterns are related and we demonstrate how a simpler patterns can be transformed into a more complex one.

All these three patterns were realised in the tool and are available from [63].

The next example discusses a simple refinement pattern for data refinement. We discuss some limitations of our approach in relation to handling expression rewriting.

We also present a procedure for constructing design component integrating third-party, off-the-shelf software components into formal developments. This example opens a number of questions for further research in this area.

The chapter is concluded with a summary of the tool for working with refinement patterns in the RODIN Event-B Platform.

6.2 N-version Programming

N-Version Programming is a software engineering method for tolerating mistakes in software implementation by using a number of functionally-equivalent versions developed independently according to common requirements or specifications [95]. The method is based on selecting the majority result from the outputs of all the versions (Figure 6.1).

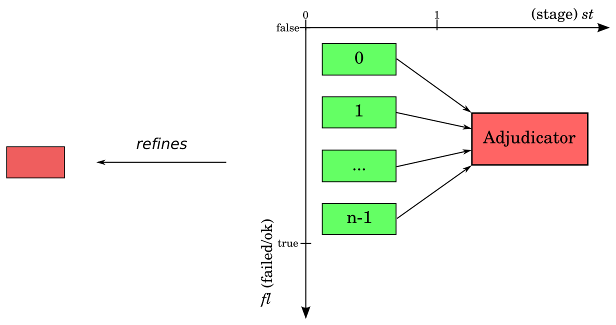


Figure 6.1: The NVP pattern. Versions are modelled as new events and the *adjudicator* refines the abstract behaviour event.

Our NVP pattern takes two arguments - an event b and number of blocks (i.e. versions) n. The result of the pattern application is a set of n behaviour blocks and the adjudicator which refines b.

```
pattern $nvp$
   req_$typing$ $b \in evt \wedge n \in \mathbb{N}$
   req_$grtone$ $n > 1$
```

Variable st defines the major state evolution stage of a system produced by the pattern: 0 is for collecting results from individual version, 1 for voting and 2 when the final result is available. The pattern introduces a Boolean variable fl indicating inability to find a majority.

```
variable $st$                      variable $fl$
   invariant $st \in 0..2$            invariant $fl \in \mathbb{B}$
   action $st := 0$                   action $fl := FALSE$
```

All the n versions produce their results independently and thus they must operate on disjoint state spaces. A simple solution is to introduce a function from a block id into a state associated with the block and let each block modify its own state using the function. Such approach, however, results in several unattractive properties of the pattern. Firstly, it introduces a variable shared by all the blocks - the state function variable - and there is nothing preventing an unadvised designer from accidentally mixing block states, both in a model and in implementation. It also prohibits an automated refinement into an efficient, concurrent implementation. Secondly, and more importantly, a refinement produced by such pattern may not easily legible. State of a block is likely to be a complex data type. Dealing with such involved structures is much more than with individual variables: proof obligations and pattern rules become rather bulky. Hence, we need to introduce a separate set of variables for each behaviour block. This complicates the pattern definition and makes the pattern correctness proofs slightly more difficult but the result is a refinement pattern which easy to use. After applying it, a designer gets n new events which are similar to the abstract event to which the refinement pattern is applied.

Each behaviour block has a Boolean variable attached indicating that the block has finished and the voter event can use the current result. This variable can also be used to disable permanently faulty blocks, although we do not do this in the current version to keep the pattern general.

The body of a behaviour block is almost an exact copy of event b with an addition of the actions assigning values to copies of the original abstract variables. Each behaviour block has its own set of copied variables.

```
forall $i$ where
    $i \in 1..n$
do
    variable $rd$
        label rd_$i$
        invariant $rd \in \mathbb{B}$
        action $rd := FALSE$
    event $alt$
        label alt_$i$
        guard $st = 0$
        guard $rd = FALSE$
        forall $a$ where
            $a \in b.\text{actions}$
        do
            variable $cp$
            label $a.\text{variable.name\_}i$
            invariant $cp \in a.\text{variable.type}$
            action $cp$ $a.\text{variable.init.style}$ $a.\text{variable.init.expr}$
            action $cp$ $a.\text{style}a.\text{expr}$
        end
        action $rd := TRUE$
end
```

When the results from all the blocks are available, the voter can compute the final result. To produce a scalable solution we have to aggregate individual variables used in different blocks into a single variable, which is a function from a block id into the block state. We do not expect designers to change this part of the specification thus we are free to use the most suitable approach here.

```
variable $rs$ for $nvp$
   label $b$_result
   invariant $rs : \mathbb{N} \hookrightarrow (\bigtimes_{a \in b.\text{actions}} a.\text{variable.type})$
   action $rs := \varnothing$
```

where $(\bigtimes_{a \in b.\text{actions}} a.\text{variable.type})$ is the type of a tuple $(v_1, v_2, \ldots v_n)$ used to store all the variable assigned in event b.

The following event constructs the function of results from the result of the individual blocks

```
event $accum$
   label $b$_collect
   guard $st = 0$
   guard $\bigwedge_{i \in 1..n}$ rd_$i = TRUE$
   action $rs := \bigcup_{i \in 1..n} \{i \mapsto (\mapsto_{a \in b.\text{actions}} \; var\_i)\}$
   action $st := 1$
```

where $var\_i = a.\text{variable.name\_}i$.

The adjudicator event refines abstract event b. The pattern adds new guards, parameters and actions and also changes the abstract action. The parameters are used as local variables which help to select the final majority result. The event guard describes a simple voting protocol and there is an action indicating if the winning result has got the majority of votes.

Parameter k is the index of the winning result, parameters $a.\text{variable.name\_}\mathtt{t}$ are used to extract solution from function rs.

```
variable $k$ for $b$
   invariant $k \in dom(rs)$
forall $a$ where
   $a \in b.\text{actions}$
do

   variable $t$ for $b$
      $a.\text{variable.name\_}\mathtt{t}$
      invariant $t \in a.\text{variable.type}$
end
```

The first guard makes the event enabled at stage 1, the next one selects k such that k is an index of a winning solution (k is the index of a winning solution if for all j different from k the number of indices pointing at the same solution as k is greater or equal to the number of solutions pointed to by j) and the last guard binds parameters to the values of the solution.

```
guard $st = 1$ for $b$
guard $\forall j \cdot (j \in dom(rs) \land j \neq k \Rightarrow$
        $card(rs^{-1}[\{rs(k)\}]) \geq card(rs^{-1}[\{rs(j)\}]))$ for $b$
guard $(\mapsto_{a \in b.\text{actions}}\ a.\text{variable.name\_t}) = rs(k)$ for $b$
```

In the event body an abstract action is replaced with an action copying values from the parameters used to extract the solution. The stage variable is advanced to indicate that the final result is available and for all the blocks the status variable is to *false* to prepare for a possible next iteration.

```
forall $a$ where
    $a \in b.\text{actions}$
do

    action $a.\text{variable.name} := a.\text{variable.name\_t}$ for $b$
end
action $st := 2$ for $b$
forall $i$ where
    $i \in 1..n$
do
    action $\texttt{rd\_}i := FALSE$ for $b$
end
action $fl := bool(card(rs^{-1}[\{rs(k)\}]) < (n/2 + 1))$ for $b$
```

Here fl is a Boolean flag indicating whether the solution has got the majority of votes or not.

6.2.1 NVP Pattern Correctness

Most proof obligations for this pattern are trivially discharged and the techniques employed are the same as those used for the Recovery Block pattern.

The only non-trivial part is to demonstrate that the voting event refines the abstract behaviour event. In other words, a solution selected by the voting event must satisfy the specification of the abstract event b. However, since the pattern does not itself produce diverse version blocks and further refinements of version blocks satisfy the abstract event b specification by definition of refinement, the voting mechanism has no effect on the selection of the result. It is enough to demonstrate that the values carried through the rs function are the results collected from version blocks. It is obviously so, since the function is only assigned in the event *accum*, which copies version results.

6.3 Parity and Hamming Patterns

In this section we consider a design of a distributed system with a large number of not completely reliable interconnections. We produce a reusable pattern that uses the parity check bit algorithm to detect possible transmission errors. Then we review the solution to produce a refined version of the pattern implementing a single error correcting algorithm. With this example we demonstrate that the pattern approach can comfortably handle complex design solutions and the patterns produced during the development are valuable and reusable.

6.3.1 Abstract Model

In its simplest, a communication loop is modelled by allocating a shared variable for a communication channel. To indicate that a channel has data to be read, the sender raises a ready flag. In its turn, the receiver, after reading from the channel, resets the flag to allow the sender to send new data. The process is repeated until the sender does not have any data to send or the receiver does not want to read the incoming data anymore. All this functionality is described by the following abstract model:

```
SYSTEM $SendRecv$
VARIABLES
    $ch, outv, rd$
INVARIANT
    $ch \in 0\,..\,15$
    $outv \in 0\,..\,15$
    $rd \in \mathbb{B}$
    $rd = false \Rightarrow outv = ch$
INITIALISATION
    $ch := 0$
    $outv := 0$
    $rd := false$

$send$ = ANY $v$ WHERE
        $rd = false \wedge v \in 0\,..\,15$
    THEN
        $ch := v$
        $rd := true$
    END

$receive$ = WHEN
        $rd = true$
    THEN
        $outv := ch$
        $rd := false$
    END
```

We assume that the model is a part of a larger model and there can be additional conditions for the event guards and a more interesting way of generating output data on the sender side. Variable rd is channel status flag. It is set to truth when the channel contains fresh data. The $outv$ is used to here to indicate that the value of the channel variable ch is accessed by the receiver.

6.3.2 Parity Check Pattern

To cope with unreliable communication channels we add redundant information to each communication message. The simplest form of this approach is to add a parity bit that is set in such a way that the sum of all bits is even (or odd, but this must be agreed upon beforehand). Single parity bit can detect single error in transmission (Figure 6.2). It is likely to give false negatives if more than one error occurs.

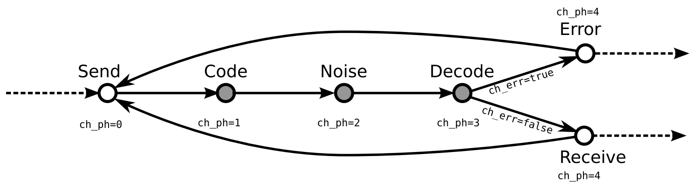


Figure 6.2: Communication loop with error detection.

The refinement pattern introduces three new variables: phase variable, controlling the order of the new events, error flag raised when parity check fails at the receiver side and the new representation of the abstract channel variable.

The table below describes how the state of the phase variables translates into system stages. In addition, the error flag variable selects between normal receiver event and an error handler. The tables below illustrate the major stages of the parity check bit algorithm and demonstrate the binary encoding with the parity check bit.

send	$ch\_ph = 0$
$\underline{code}$	$ch\_ph = 1$
$\underline{noise}$	$ch\_ph = 2$
$\underline{decode}$	$ch\_ph = 3$
receive or $\underline{ch\_error}$	$ch\_ph = 4$

decimal	binary with parity bit
0	$0\,0\,0\,0\,\underline{0}$
1	$1\,0\,0\,0\,\underline{1}$
2	$0\,1\,0\,0\,\underline{1}$
3	$1\,1\,0\,0\,\underline{0}$
15	$1\,1\,1\,1\,\underline{0}$

The new channel variable is a bit-wise representation of the abstract channel. It is defined in such a way that it has enough bits to encode any value that can be stored by the abstract channel and also the parity check bit. The pattern has three parameters - abstract channel variable, abstract sender and abstract receiver:

pattern *parity*
  req_*typing* $ch \in var \wedge snd \in evt \wedge rcv \in evt$
  req_*channel* $ch.\mathrm{typ} \subseteq \mathrm{finite}(\mathbb{Z}) \wedge ch.\mathrm{typ} \neq \emptyset$
  req_*attr* $ch.\mathrm{class} = \texttt{channel} \wedge snd.\mathrm{sender} = ch \wedge rcv.\mathrm{receiver} = ch$

The req_*typing* requirement types the parameters, in requirement req_*channel* we state that a channel variable has an integer type and can accept at least one value. Finally, we use model annotations to make sure that the selected events and the channel variable have the same interpretation in the model and in the pattern.

To simplify pattern definition, several constants are defined. Constant mxx is a zero-adjusted maximum integer number that can be placed in the abstract channel, zdj is the low boundary for the channel type, it is used to convert zero-adjusted binary encoding back into decimal integer. The minimal number of bits required to encode a number is calculated as $\lg(n) + 1$, this value is stored in constant dbt. Finally, nch is the number of check bits, always 1 for the parity bit pattern, and tbt is the total number of bits:

$$\begin{aligned} mxx &= \max(ch.\text{type}) - \min(ch.\text{type}) \\ zdj &= \min(ch.\text{type}) \\ dbt &= \lfloor \lg(mxx) \rfloor + 1 \\ nch &= 1 \\ tbt &= dbt + nch \end{aligned}$$

The channel variable is a sequence of bits. It is initialised to be all zeroes which corresponds to zero in the decimal integer notation. Here we rely on knowledge that the abstract model also uses zero to initialise the channel variable:

variable b
  invariant $b \in 1..tbt \rightarrow \{0, 1\}$
  action $b := 1..tbt \times \{0\}$

Phase and error flags are new model variables. The ph variable represents the stages of the algorithm:

variable ph
  invariant $ph \in 0..4$
  action $ph := 0$

The err variable is a flag indicating a failed transmission:

variable err
  invariant $err \in \mathbb{B}$
  action $err := false$

Before we start defining the main part of the pattern, it is convenient to formulate the desired pattern properties. In this case we state that just before a bit is flipped during transmission, the new channel variable is exactly the same as the old one. By this we check that the encoding pattern is correct. The other property states that if the parity check was successful, then the received value is the one that was sent. We assume here that the hamming distance between channel values before bit-flip and after is at most one, i.e. at most one bit is changed:

```
**invariant** $icod$
    **expression** $ph = 2 \Rightarrow ch = zdj + \sum_{i \in 1..dbt} 2^{i-1} * b(i)$
**invariant** $idec$
    **expression** $ph = 4 \wedge err = false \Rightarrow ch = zdj + \sum_{i \in 1..dbt} 2^{i-1} * b(i)$
```

The coding event computes a binary representation of a value to be sent through the channel and adds a parity check bit as well. This is done by declaring event parameters, one per each bit, in the concrete channel variable and stating that they are such that when converted into decimal form yield the value of the abstract channel. The parity bit value is the sum of all bits modulo 2:

```
event $code$
    param$_{i \in 1..dbt} bit_i \in \{0,1\}$
    param $p \in \{0,1\}$
    guard $ph = 1$
    guard $ch = zdj + \sum_{i \in 1..dbt} 2^{i-1} * bit_i$
    guard $p = \sum_{i \in 1..dbt} bit_i \bmod 2$
    action $b := \bigcup_{i \in 1..dbt} \{i \mapsto bit_i\} \cup$
                      $\{dbt + 1 \mapsto p\}$
    action $ph := 2$
```

The decoding event checks that the transmitted value is correct and sets the error flag accordingly:

```
event $decode$
    param $par \in \{0,1\}$
    guard $ph = 3$
    guard $par = \sum_{i \in 1..tbt} b(i) \bmod 2$
    action $err := par \neq 0$
    action $ph := 4$
```

The noise event random a bit of value in the communication channel. In some cases this changes the channel value, in other it does not:

```
event $noise$
    param $bit \in 1..tbt$
    param $val \in \{0, 1\}$
    guard $ph = 2$
    action $b := b \lhd\!\!\!+ \{bit \mapsto val\}$
    action $ph := 3$
```

The error event refines the abstract receiver event by extending it with the new guards and action. An error handling mechanism can be introduced by refining this event:

```
event $error$
    refines $rcv$
    variable $rcv$.variables
    guard $ph = 4$
    guard $err = true$
    guard $rcv$.guard
    action $ph := 0$
    action $rcv$.actions
```

The pattern adds new guards and actions to the sender and receiver events to enforce the ordering of new events. The sender event initiates the chain of events. This event is activated at the phase zero and passes control to the next event by incrementing the phase variable:

guard $ph = 0$ **for** snd
action $ph := 1$ **for** snd

The receiver event concludes the event chain. It is responsible for passing control back to the sender event:

guard $ph = 4$ **for** rcv
guard $err = false$ **for** rcv
action $ph := 0$ **for** rcv

Most correctness conditions for the pattern are very simple. The only really non-trivial condition requests us to show that when the parity check claims undistorted transmission the decoded value is indeed the original value sent by the sender:

$$par = 0 \land par = \sum_{i \in 1..tbt} b(i) \bmod 2 \Rightarrow ch = zdj + \sum_{i \in 1..dbt} 2^{i-1} * b(i)$$

The core of the proof is the deduction of the fact that the hamming distance between channel values before and after the noise event is at most one. This can be done by analysing the specification of the noise event. Also, it is not hard to see that the pattern is monotonic as it only adds new elements and only refers to the abstract elements in the invariant.

An example of a model produced by this pattern can be found in Appendix C.5.

6.3.3 Hamming Coding Pattern

In this section we discuss a refinement of the parity pattern which automatically corrects transmission errors.

Since our noise event flips only one bit, we can do much better than simply adding a parity check. A single error correcting code, such as the Hamming codeword [96], can tolerate change in a single bit of a transmitted word by adding several check bits (Figure 6.3). The general idea behind the Hamming coding algorithm is placing enough check bits in such manner that it is possible to find which bit was flipped. The table below contains examples of hamming code words with the check bits underlined.

decimal	hamming coding (7, 4)
0	$\underline{0}\,\underline{0}\,0\,\underline{0}\,0\,0\,0$
1	$\underline{1}\,\underline{1}\,1\,\underline{0}\,0\,0\,0$
2	$\underline{1}\,\underline{0}\,0\,\underline{1}\,1\,0\,0$
3	$\underline{1}\,\underline{1}\,1\,\underline{1}\,1\,0\,0$
15	$\underline{1}\,\underline{1}\,1\,\underline{1}\,1\,1\,1$

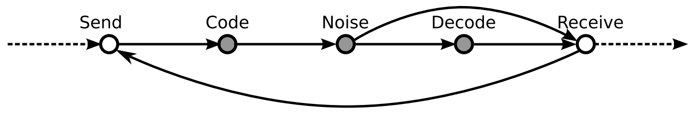


Figure 6.3: Communication loop with error correction.

There must be enough check bits to encode position of any data bit. Hence, for a channel transmitting a value made of $hdbt$ bits, the required number of check bits is

$$hnch = \quad \lfloor \lg(hdbt) \rfloor + 1$$

For other constants, we reuse the definitions from the parity check bit pattern, adding prefix h to the name of each constant. To simplify further discussion we introduce two helper functions. The first one maps a serial position of data bit code into its position in a hamming codeword:

$$\begin{array}{l} dsq \in 1..tbt \rightarrowtail\!\!\!\!\!\rightarrow 1..tbt \setminus \{2^{k-1} \mid k \in 1..nch\} \\ \forall i \cdot (i \in 2..tbt \Rightarrow dsq(i) > dsq(i-1)) \end{array}$$

The few initial values of dsq are $dsq(1) = 3; dsq(2) = 5; dsq(3) = 6; dsq(4) = 7; dsq(5) = 9$. Hence, the first data bit can be found in position 3, the second in position 5 and so on. The second constant maps a serial of a check bit code into a set of codeword bits it checks (check bit checks other check bits):

$$\begin{array}{l} cst \in 1..nch \rightarrowtail\!\!\!\!\!\rightarrow \mathcal{P}(1..tbt) \\ cst = \{\{i \mapsto s\} \mid i \in 1..nch \wedge s = \{k \mid k \in i+1...tbt \wedge k \otimes 2^{i-1} \neq 0\}\} \end{array}$$

The sample values of the function are $cst(1) = \langle 1, 3, 5, 7, 9, 11, 13, 15, 17 \ldots \rangle$, $cst(2) = \langle 2, 3, 6, 7, 10, 11, 14, 15 \ldots \rangle$, $cst(3) = \langle 4, 5, 6, 7, 12, 13, 14, 15 \ldots \rangle$.

The essential property of a hamming codeword is that if parity check fails for some check bit then number $\sum_{i \in 1..nch} 2^{i-1} * ((\sum_{j \in cst(i)} b_j) \bmod 2)$ (which is always non-zero when a check fails) indicates the bit position which was flipped in transition. Here $(\sum_{j \in cst(i)} b_j) \bmod 2$ is checksum for parity bit i and the outer summation converts from binary representation of a bit position encoded in values of parity checksums. This works due to the way check bits are arranged in a codeword. The set $cst(n)$ is made of numbers that binary representation contains 1 in n-th position. If a bit from set $cst(k)$ is flipped then k-th checksum fails indicating that one of these bits is bad. But other checksums will fail too, and the intersection of all such sets would result in exactly one number which evaluates to the position of the bit which was changed. We calculate the intersection with the outer summation in the first formula.

The new pattern has the same set of parameters. In the pattern declaration we also state the pattern refines pattern *parity*:

```
pattern $hamming$
  req_$typing$ $hch \in Var \wedge hsnd \in evt \wedge hrcv \in evt$
  req_$channel$ $hch.\mathrm{typ} \subseteq \mathrm{finite}(\mathbb{Z}) \wedge hch.\mathrm{typ} \neq \varnothing$
  req_$attr$ $hch.\mathrm{class} = \mathtt{channel} \wedge hsnd.\mathrm{sender} = hch \wedge hrcv.\mathrm{receiver} = hch$
```

The gluing invariants, needed to demonstrate the refinement relation between the patterns, link variables of the hamming pattern with variables the parity pattern:

$$\begin{cases} hph = ph \\ hch = ch \\ ph = 2 \implies zdj + \sum_{i \in 1..dbt} 2^{i-1} * b(i) = \\ \qquad\qquad hzdj + \sum_{i \in 1..hdbt} 2^{i-1} * hb(dsq(i)) \\ ph = 4 \wedge err = \mathit{false} \implies zdj + \sum_{i \in 1..dbt} 2^{i-1} * b(i) = \\ \qquad\qquad hzdj + \sum_{i \in 1..hdbt} 2^{i-1} * b(dsq(i)) \end{cases}$$

The first invariant says that hp and hph are the same variables, the second equates the channel variables. The *coding* invariant states how the states of the patterns are linked after the *code* event. The *decoding* event relates states after error detection and error correction. It says that if there were no errors detected in the parity pattern the result is the same as in the hamming. The invariant underlines the value of the new pattern - it always produces a correct result while the parity pattern only occasionally.

The invariants are similar to those of the parity pattern. After decoding, the data bits of a codeword must be the same as the value of the abstract channel. After decoding, the value must be same as in the perfect abstract channel. The major difference here is that we are able to state that a transmission error can never occur. Strictly speaking, these invariants are not needed to demonstrate the pattern correctness as they can be deduced from the gluing invariants:

```
invariant $icod$
  expression $ph = 2 \Rightarrow ch = zdj + \sum_{i \in 1..dbt} 2^{i-1} * b(dsq(i))$
invariant $idec$
  expression $ph = 4 \Rightarrow ch = zdj + \sum_{i \in 1..dbt} 2^{i-1} * b(dsq(i))$
```

The coding event computes a binary representation of a value sent over the channel and also computes the values of check bits. The result is a channel value which is a hamming code word for a given input value:

```
event $code$
  guard $ph = 1$
  param$_{i \in ran(dsq)}$ $bit_i \in \{0, 1\}$
  guard $ch = zdj + \sum_{i \in 1..dbt} 2^{i-1} * bit_{dsq(i)}$
  action $b := \bigcup_{i \in 1..dbt}\{dsq(i) \mapsto bit_{dsq(i)}\} \cup \bigcup_{i \in 1..nch}\{2^{i-1} \mapsto \sum_{j \in cst(i)} bit_j\}$
  action $ph := 2$
```

The decoding event models the error correction part of the hamming coding algorithm. This event detects whether any parity check bit has failed and reconstructs the original codeword:

```
event $decode$
   guard $ph = 3$
   variable $_{i \in 1..nch}\ par_i \in \{0,1\}$
   guard $_{i \in 1..nch}\ par_i = \sum_{j \in cst(i)} b_j \bmod 2$
   guard $_{i \in 1..nch}\ par_i \neq 0$
   action $b := b \Leftarrow \{\sum_{i \in 1..nch} 2^{i-1} * par_i \mapsto (b(\sum_{i \in 1..nch} 2^{i-1} * par_i) + 1) \bmod 2\}$
   action $ph := 4$
```

The pass-through event handles the case when correction is needed. It advances the phase variable without changing the channel variable:

```
event $passthru$
   guard $ph = 3$
   variable $_{i \in 1..nch} par_i \in \{0,1\}$
   guard $_{i \in 1..nch} par_i = \sum_{j \in cst(i)} b_j \bmod 2$
   guard $_{i \in 1..nch} par_i = 0$
   action $ph := 4$
```

The refinement of the abstract sender and receiver events are exactly the same as in the parity pattern.

The pattern has one non-trivial proof obligation. It is needed to show that after decoding the resulting value is exactly the same as the value sent initially by the sender. The proof essentially requires a careful analysis of the whole algorithm. For this we have relied on the original algorithm description [96]. It is out of question that such can be automatically handled by the theorem prover. We still believe that construction of such proof using Event-B interactive prover might be possible but is bound to be very difficult.

A sample for this pattern is presented in Appendix C.6.

6.4 Characteristic Function Pattern

Transformation of an abstract variable type into a more efficient concrete type is an important case of data refinement. This step is almost universally required when an abstract model is detalised to produce an implementation. In this section we consider one example of such refinement where an abstract set variable is transformed into a characteristic function variable, that is, instead of an abstract variable $set \in \mathcal{P}(X)$, a refined model uses new concrete $chf : X \nrightarrow \mathbb{B}$ and the gluing invariant is $set = chf^{-1}[\{true\}]$.

This example also serves as a demonstration of the limitations of the pattern language. While other examples were mainly concerned with incremental, structural changes, this one requires rather fine level of changes in the existing model elements. The main difficulty is in developing patterns transforming expressions to realise a data refinement.

A refinement pattern implementing this data refinement step accepts a model with a set variable and transforms it into a characteristic function, rewriting various model elements. Model invariants and event guards are refined by replacing all occurences of the abstract set variable with the expression relating to the characteristic function

$$I(set, v) = I(chf, v)[chf^{-1}[\{true\}]/set]$$

Deterministic actions are refined by first eliminating the abstract set variable and then constructing a characteristic function by mapping the result (new set elements) into truth

$$\begin{array}{l} set := F(set, v) \\ set := (F(chf, v)[chf^{-1}[\{true\}]/set]) \times \{true\} \end{array}$$

A refinement rule constructing this transformation by substituting variable set in the list of assigned variables with chf and rewriting the expression looks like this:

$$\begin{array}{l} \mathsf{newactionvar}^{-1}(a, set) \\ \quad \mathsf{newactionvar}(a, chf) \\ \quad \mathsf{newactionexp}(a, (a.\exp[chf^{-1}[\{true\}]/set]) \times \{true\}) \end{array}$$

Non-deterministic "belongs to" actions are refined in a similar way: a characteristic function is selected among the elements of the cartesian product of the original set and the power set of $true$

$$\begin{array}{l} set :\in F(set, v) \\ chf :\in (F(chf, v)[chf^{-1}[\{true\}]/set]) \times \mathcal{P}(\{true\}) \end{array}$$

The corresponding pattern rule is

$$\begin{array}{l} \mathsf{newactionvar}^{-1}(a, set) \\ \quad \mathsf{newactionvar}(a, chf) \\ \quad \mathsf{newactionexp}(a, (a.\exp[chf^{-1}[\{true\}]/set]) \times \mathcal{P}(\{true\})) \end{array}$$

Finally, the general form of a non-deterministic action is refined by simply rewriting the constraining predicate

$$
\begin{array}{l}
set :| F(set, v) \\
chf :| F(chf, v)[chf^{-1}[\{true\}]/set]
\end{array}
$$

The same approach is used to refine non-deterministic actions constraining multiple variables

$$
\begin{array}{l}
set, u :| F(set, v, u) \\
chf, u :| F(chf, v, u)[chf^{-1}[\{true\}]/set]
\end{array}
$$

And the pattern rule for these two cases is

```
newactionvar$^{-1}(a, set)$
  newactionvar$(a, chf)$
  newactionexp$(a, a.\exp[chf^{-1}[\{true\}]/set])$
```

Now we can formulate a general rule rewriting a model action:

```
$act(a, s, c) =$
(
when
    $s \in a.\mathrm{var}$
do
    newactionvar$^{-1}(a, s)$
      when
          $a.\mathrm{sty} = (:=)$
      do
          newactionvar$(a, c)$
          newactionexp$(a, (a.\exp[c^{-1}[\{true\}]/s]) \times \{true\})$
      end
      when
          $a.\mathrm{sty} = (:\in)$
      do
          newactionvar$(a, c)$
          newactionexp$(a, (a.\exp[c^{-1}[\{true\}]/s]) \times \mathcal{P}(\{true\}))$
      end
      when
          $a.\mathrm{sty} = (:|)$
      do
          newactionvar$(a, c)$
          newactionexp$(a, a.\exp[c^{-1}[\{true\}]/s])$
      end
end
)
```

This rule can applied to an action assigning to the abstract variable set. Now we are able to formulate the complete refinement pattern. The pattern accepts a single parameter which is a set variable

```
pattern $set2chf$

    req_$setvar$ $set \in var \wedge isset(set)$
```

The set variable is not used in the concrete model and is removed from the list of the system variables:

$$\mathsf{addvar}^{-1}(set)$$

New variable chf is defined as a function mapping elements of the abstract set variable into booleans:

```
addvar($chf$)
   newvartype($chf, (basetype(set) \to \mathbb{B})$)
   newvaraction($v, set.\mathrm{act}$)          $act(set.\mathrm{act}, set, chf)$
```

The gluing invariant links the concrete chf variable with the abstract set variable:

$$\textbf{invariant } set = chf^{-1}[\{true\}]$$

Model invariants are rewritten by replacing set with $chf^{-1}[\{true\}]$. For this we first remove an abstract invariant and add the updated version of the same invariant

```
forall $i$ where
    $i \in inv$
do
    addinv$^{-1}(i)$
       addinv$(e, i[chf^{-1}[\{true\}]/set])$
end
```

To refine model events, we have to rewrite event guards and refine the event actions. It is convenient to split this task into three sub-rules: rewriting event guards, rewriting expressions of actions not assigning to set and rewriting actions updating set

```
forall $e$ where
    $e \in evt$
do
    $grd(e) expr(e)$
    $subs(e)$
end
```

Event guard is replaced with a new version which uses $chf^{-1}[\{true\}]$ everywhere instead of set:

$$grd(e) = \left(\begin{array}{l} \textsf{forall } g \textsf{ where} \\ \quad g \in e.\mathrm{grd} \\ \textsf{do} \\ \quad \mathsf{addguard}^{-1}(e, g) \\ \qquad \mathsf{addguard}(e, g[chf^{-1}[\{true\}]/set]) \\ \textsf{end} \end{array} \right)$$

Action expressions are treated in the same manner:

$$expr(e) = \left(\begin{array}{l} \textsf{forall } a \textsf{ where} \\ \quad a \in e.\mathrm{act} \wedge a.\mathrm{var} \neq \{set\} \\ \textsf{do} \\ \quad \mathsf{newactionexp}(a, a.\mathrm{exp}[chf^{-1}[\{true\}]/set]) \\ \textsf{end} \end{array} \right)$$

Finally, all the updates to the set variable are transformed using the act action rewrite rule defined above:

$$subs(e) = \left(\begin{array}{l} \textsf{forall } a \textsf{ where} \\ \quad a \in e.\mathrm{act} \wedge a.\mathrm{var} = \{set\} \\ \textsf{do} \\ \quad act(a, set, chf) \\ \textsf{end} \end{array} \right)$$

The pattern is demonstrated on the following simple model. The model defines a single variable, which is a set of natural numbers, and two events, each transforming the variable in different ways

SYSTEM set

VARIABLES set

INVARIANT $set \subseteq \mathbb{N}$

INITIALISATION $set := \varnothing$

EVENTS

add = **ANY** e **WHERE** $e \in \mathbb{N} \setminus set$ **THEN** $set := set \cup \{e\}$ **END**

del = **ANY** e **WHERE** $e \in set$ **THEN** $set := set \setminus \{e\}$ **END**

Applying our refinement pattern we obtain the following refinement of the *set* model

```
SYSTEM $chf$
REFINES $set$
VARIABLES $chf$
INVARIANT
    $chf : \mathbb{N} \pfun \mathbb{B}$
    $set = chf^{-1}[\{true\}]$
INITIALISATION $chf := \varnothing$
EVENTS
    $add$  $=$  ANY $e$ WHERE
                  $e \in \mathbb{N} \setminus chf^{-1}[\{true\}]$
                THEN
                  $chf := (chf^{-1}[\{true\}] \cup \{e\}) \times \{true\}$
                END
    $del$  $=$  ANY $e$ WHERE
                  $e \in chf^{-1}[\{true\}]$
                THEN
                  $chf := (chf^{-1}[\{true\}] \setminus \{e\}) \times \{true\}$
                END
```

The result, however, is not completely satisfactory. Although technically the use of the *set* is now avoided and the abstract set is described using a characteristic function, the refined model provides only some of the intended benefits of the data refinement and is far from being legible or efficient. Ideally, a pattern of this kind should be able to construct something like the model below:

```
SYSTEM $chf$
REFINES $set$
VARIABLES $chf$
INVARIANT
    $chf : \mathbb{N} \pfun \mathbb{B}$
    $set = chf^{-1}[\{true\}]$
INITIALISATION $chf := \varnothing$
EVENTS
    $add$  $=$  ANY $e$ WHERE $chf(e) = false$ THEN $chf := chf \mathbin{<\!\!+} \{e \mapsto true\}$ END
    $del$  $=$  ANY $e$ WHERE $chf(e) = true$ THEN $chf := chf \mathbin{<\!\!+} \{e \mapsto false\}$ END
```

This is not possible with the current pattern language since the means for transforming expression are very rudimentary and there is no template matching mechanism for expressions. Besides, we would have to define a much more elaborate pattern caring for various use cases a set variable in different contexts. Such pattern should be able to recognise the role of a set variable and select the most appropriate characteristic function equivalent, e.g. $chf(e) = true$ for $e \in set$, but $chf^{-1}[\{true\}] \cup a$ for $set \cup a$. In additional, the same abstract expression should be rewritten in different ways to provide the most efficient implementation for different prospective implementation languages. To implement such transformation we plan to extend the pattern language with a flexible rewriting mechanism similar to those used in theorem provers.

6.5 Design Component and Software Components

This is a small example illustrating how off-the-shelf components can be integrated into a formal development using the design component concept. We assume that a software component is constructed formally and intermediate development results are available. We require that a component is constructed in a special manner. At some point of development, the model of the component must specifies an abstraction of a component environment together with an abstraction of a component behaviour. At this stage, the model does not yet differentiate between the functionality of environment and the services provided by the component. We call this model an *integration point* and denote the class of such models as $integpnt$ (the model class here is used in a sense of input/output model class of transformation; each component development would use its own definition of $integpnt$).

The development proceeds further until a model clearly differentiates between the environment and the component parts. We denote corresponding model classes as env and $component$ and the model class for the combination of these two is $env * component$. Here the $*$ operator requires that env and $component$ hold at the same time but are defined on disjoint model parts. In other words, the environment and component parts of a model may not share model elements (the complete model, however, may contain additional elements used to glue these two parts).

After this point of a development, we assume that an environment specification and the way the component interacts with it are fixed and only the component functionality is refined further. This process stops when a component implementation is constructed. The final model of a component is referred to as $cmp\_implementation$.

The development tactics we have just described corresponds to a modelling pattern of the following form

```
from $true$ achieve $integpnt$
  then from $integpnt$ achieve $env * component$
    then maintain $env$ and from $component$ achieve $cmp\_implementation$
```

Each component development defines different assumptions and goals although the overall pattern structure remains the same. A development based on this modelling pattern has the following structure

```
$mk\_integpnt$
  then $mk\_interface$
    then $mk\_implementation$
```

where refinement pattern $mk\_integpnt$ is an implementation of from $true$ achieve $integpnt$, and $mk\_interface$ and $mk_implementation$ are refinement patterns implementing the corresponding parts of the modelling pattern. To integrate a component into a development, a modeller uses the following design component

```
from $true$ achieve $integpnt$
  then $mk\_interface$
    then maintain $component$
      then $mk\_implementation$
```

The from $true$ achieve $integpnt$ states the integration requirements of a pattern: in order to use a pattern the development must arrive to a model satisfying the $integpnt$ predicate. Once this is accomplished, the $mk\_interface$ refinement pattern is used to construct a refinement based on the $env * component$. Then a modeller is free do some arbitrary refinements steps (based on other modelling patterns, of course). It is up to the modeller to decide when to stop and use the $mk\_implementation$ pattern incorporating the complete final model of the component into the development.

We can also use a plain refinement chain as a basis for an integration design component. A result of a component development is a refinement chain of the following form

$$s_0 \sqsubseteq \ldots \sqsubseteq s_n \sqsubseteq \ldots \sqsubseteq s^i_1; s^c_1 \sqsubseteq \ldots \sqsubseteq s^i_1; s^c_k$$

where $a; b$ denotes a model composed of two parts and the following properties hold

$$\begin{array}{l} integpnt(s_n)\wedge \\ env(s^i_1) * component(s^c_1)\wedge \\ cmp\_implementation(s^c_k) \end{array}$$

Now, the design component is based on the calculation of model differences, although the general structure remains the same:

```
from $true$ achieve $integpnt$
  then $\Delta(s_n, s^i_1; s^c_1)$
    then maintain $component$
      then $\Delta(s_1; s^c_k)$
```

To integrate a third-party component into a development, a modeller activates the related modelling pattern. The first part of such pattern - from $true$ achieve $integpnt$ - requires the modeller to identify a component integration point by making the development compatible with the $integpnt$ model class. Once this is done, a full component interface and a detalised version of a component environment are automatically added to the model with the help of refinement pattern $mk\_interface$ (or model difference $\Delta(s_n, s^i_1; s^c_1)$). After this point, a modeller may not change the specification of the component interface. When the development is finished, a complete specification of a component is added using refinement pattern $mk\_implementation$ (or model difference $\Delta(s_1; s^c_k)$). This step constructs either a complete specification of a component (i.e., its implementation) or a more detalised version of the component interface needed to integrate the development with a closed-source or a remote (e.g., web-service based) component implementation.

6.6 Tool Support

Tool support is central to the pattern mechanism. Automatic instantiation of pattern is not only saving a modeller's time but also ensures that no errors are introduced and the pattern rules are strictly followed. An important effect of the tool support is that a modeller does not have to know how a pattern works, only what it does. A modeller browses through a list of available patterns, chooses a suitable pattern, applies it and reviews the result. The proof-of-the-concept implementation was constructed to validate the pattern mechanism.

The tool, called *finer* was realised as a plugin to the RODIN Event-B platform [97]). The plugin implements a subset of the pattern language and relies on the XML notation for pattern input and editing.

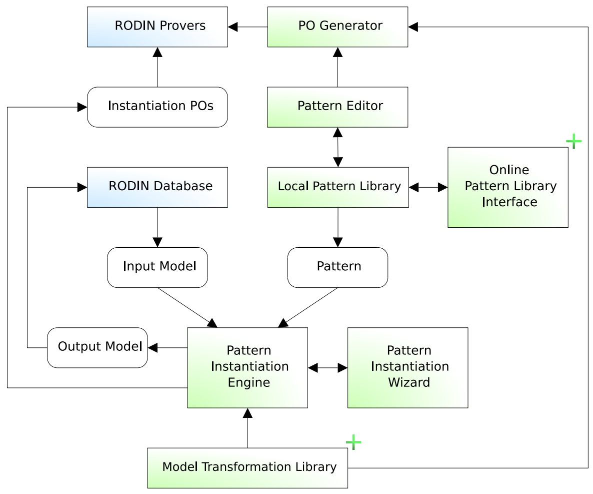


Figure 6.4: The Event-B refinement patterns tool architecture.

6.6.1 The Tool Architecture

The overall tool architecture is presented in Figure 6.4. 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. 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 tool provides basic means for pattern creation and instantiation. The main functionality of the tool is construction of a model refinement by instantiating one of the available refinement patterns. The instantiation process is made of a number of steps, as show in Figure 6.5.

First, a tool user select a model to refine from the project explorer view. The pop-up menu shows the refinement wizard option.

The plugin builds a list of patterns applicable to the current model. This is done by evaluating pattern preconditions on the model. The result is used to query a user for a desired pattern (see screen-shots in Figure 6.6).

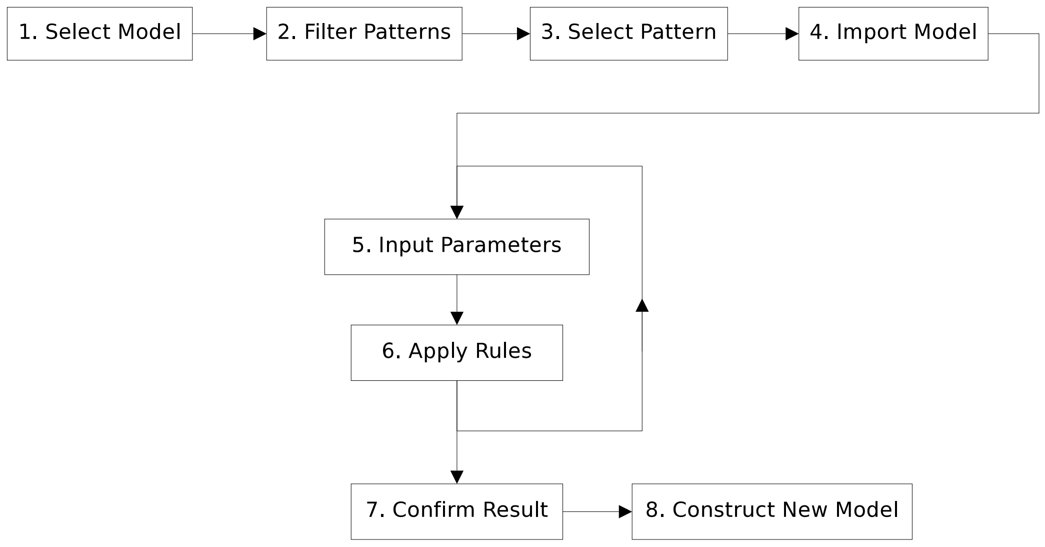


Figure 6.5: Pattern instantiation in the *finer* tool.

Once a pattern is selected, the model construction process starts. The first step is to make a copy of a model to be transformed. Later this copy is used to produce the refined model.

The selected pattern is applied to the model by evaluating the pattern rule. When a user's input is need, the wizard requests an input and then supplies the inputted values to the pattern engine code.

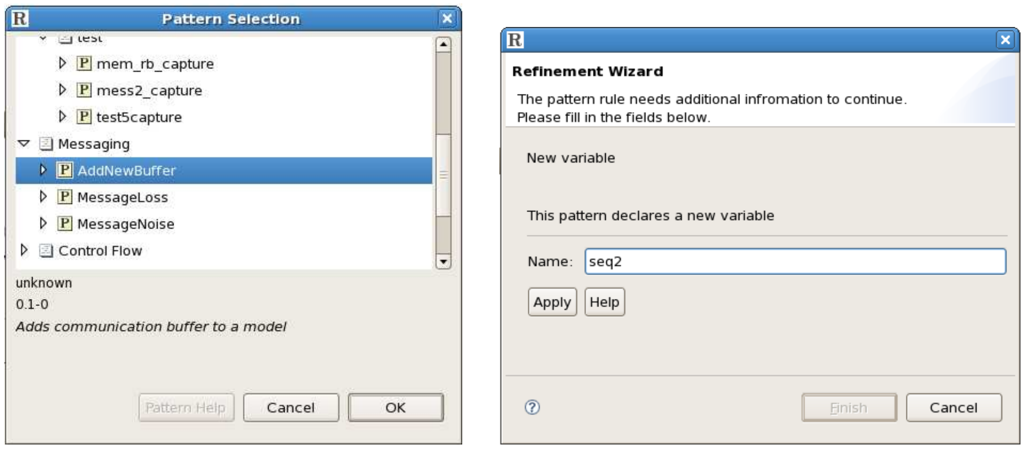
The result of a pattern evaluation is a list of instantiated model transformation rules. These rules are presented to a modeller for a review.

The last step is to apply all the model transformations rules, one after another, to the abstract model. The resulting model is saved and added to the parent Event-B project.

The result of a successful pattern instantiation is a new model and, possibly, a set of instantiation proof obligations - additional conditions that must be demonstrated each time 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.

6.6.2 Supported Pattern Language

The tool does not implement the full pattern language. The main restriction is a simplified form of the *forall* statement which supports only one free variable of a predefined type and no parameters.



test
mem_rb_capture
mess2_capture
test5capture
Messaging
AddNewBuffer
MessageLoss
MessageNoise
Control Flow

unknown
0.1-0
Adds communication buffer to a model

Refinement Wizard

The pattern rule needs additional infromation to continue.
Please fill in the fields below.

New variable

This pattern declares a new variable

Name: seq2

Apply Help

Figure 6.6: Step 3 (pattern selection) and step 5 (parameter input during pattern instantiation).

There are several forms of the *forall*-like statements for events, variables, actions, guards and so on. The corresponding XML notation is given below

`⟨event id='e'⟩`	
$r(e)$	forall e where $e \in evt$ do $r(e)$ end
`⟨/event⟩`	
`⟨variable id='v'⟩`	
$r(v)$	forall v where $v \in var$ do $r(v)$ end
`⟨/variable⟩`	
...	

Another class of statements implements the *with* statement, also with a single variable. This result in a query to a modeller to select a model element from a list of possible elements:

`⟨select:event id='e'⟩`	
$r(e)$	with e where $e \in evt$ do $r(e)$ end
`⟨/select:event⟩`	
`⟨select:variable id='v'⟩`	
$r(v)$	with v where $v \in var$ do $r(v)$ end
`⟨/select:variable⟩`	
...	

Finally, the tool language defines rules for nearly all the model transformations defined for Event-B. Some examples are given below:

```
⟨define:event id='e' name='n'⟩        event e
   r(e)                                  label n
 ⟨/define:event⟩
⟨define:guard id='g'⟩                 guard v
   grdexpr                               expression grdexpr
 ⟨/define:variable⟩
...
```

A rule can take the following arguments:

- `id='name'` - rule name. The name can be used to reference the rule from the sub-rules;
- `scope='id'` - the scope of the rule, the 'id' argument must be a name of an enclosing rule. The scope attribute defines the immediate parent rule and is used to avoid too deep nesting of patterns;
- `:name='value'` - requires that the rule contains the given annotation name/value pair;
- `predicate='expr'` - a predicate restricting rule application conditions.

The following rule is an excerpt from the Hamming pattern:

```
<select:variable id='ch'
 :class='channel'
 predicate='isint(me.type) and
typemax(me.type)-typemin(me.type) > 0'
 hint='Channel variable'>
...
```

The rule selects a variable object and names the object *ch*. Note, that this name has nothing to do with the actual variable name. The variable must carry the *class* attribute with the *channel* value. In addition, the variable must be an integer type (subset or equal to $\mathbb{Z}$) with the defined lower and upper limits which describe a non-empty value range.

The tool uses a simple expression language to evaluate conditions and dynamically construct expression. Thus, for example, $i - 1$ where i is a constant is evaluated into a constant, otherwise it is an expression with i substituted with a name of a variable it stands for:

$< define : guard >$branch=i-1$< /define : guard >$ **guard** $branch = i - 1$

The pattern language also features a number of predefined operations and functions that can be used inside the $ brackets. In addition to the common arithmetic operations, there are operations for testing equality of labels and expressions, the *belongs to* operator, string concatenation, predicates for extracting and testing variable types, string manipulations and few mathematical functions, such as $power$ and $ceil$.

Constructing a complex expression may be challenging. For example, the expanded form of $\sum_{i \in 1..nch} 2^{i-1} * par_i$ (Event-B has no Σ) has to be constructed from many pieces, all glued together by the summation operator

```
<define:formula id='sum' operator='+'>
    <index id='i' from='1' upto='$nch$'>
        <define:formula scope='sum'>
            $power(2, i-1)$*par$i$
        </define:formula>
    </index>
</define:formula>
```

The rule above works by constructing formula pieces and attaching them to the parent summation formula. When printed as a string, the outer summation could look like this: `1 *par1+2par2` (the exact expression produced by the pattern rule).

The tool implementation is freely available from our web site [63]. Once installed into the RODIN platform, it provides an environment for working with refinement patterns, i.e. for selecting, editing and applying patterns in automatic or semi-automatic manner. Thanks to the open architecture of the platform, the plugin is seamlessly integrated into the platform and provides an intuitive user interface.

The current tool version is far from being simple to use and lacks some essential functionality. In the future, the XML pattern notation will be replaced in future versions with a dialogue-based input, similar to those used for Event-B models. The notation will be changed to strictly follow the presented pattern language. This will result in a much more flexible and powerful tool. The tool will also provide means for analysing pattern correctness. Proof obligations are to be automatically generated from pattern definition and many of them can be handled by the build-in platform prover. To interface with the prover, the tool will output proof obligations as context file theorems along with free variables as context file constants. Such context file can be attached to all pattern-generated machines and the theorems can be analysed by an automatic prover.

Chapter 7

Conclusions

7.1 Summing Up

The purpose of a design component is to assist a modeller in integration of a reusable design, packed in the component, into a development the modeller works on. The inputs of a design component accept a model and a configuration. There are two types of design components: components used to develop systems using top-down refinement procedure are called *refinement components*; components used to reverse engineer a legacy system are called *abstraction components*. Consequently, the input of a refinement component is an abstract model while the input of an abstraction component is a concrete model.

Configuration is a set of parameters required to instantiate a pattern used by a component. A modeller is requested to supply values for configuration parameters just prior to a pattern instantiation.

The input and output of a design component are described by the predicates characterising the input and output model classes of the component. These predicates can be used to match two components to find whether they are composable or analyse some third-party component to see whether it can be integrated into a

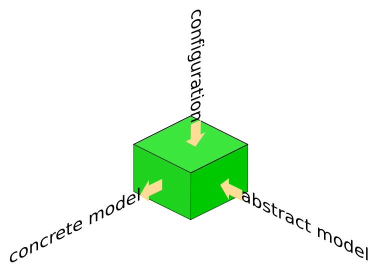


Figure 7.1: A design component is a black box with an input and output, accepting and producing models, and an additional configuration input.

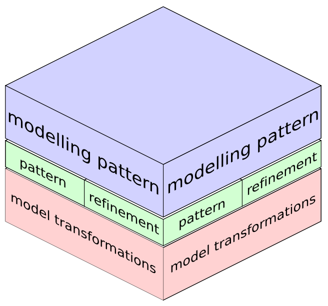


Figure 7.2: A typical design component includes a modelling pattern, a collection of a refinement patterns and a library of model transformation.

development.

Like a software component, a design component is reusable. A design component can be assembled from various elements and given to other developers. Hence, a design component is self-sufficient - it does not include links to the context in which it was created. An important property of a design component is that its application requires minimum effort from a modeller. To apply a design pattern, a modeller has to teach himself the design pattern by read the explanations and examining code example. Then this new knowledge can be transferred into modelling activity. Application of a design component is completely automated since all its parts are machine-interpretable. To use a design component, a modeller simply loads the component into the design environment and then uses a visual interface to show the step where the component should be used. From this moment, the integration of a component, which is a complex and multi-stage process, is under the control of the modelling environment.

A good design component is reusable in a wide range of contexts. For example, we can imagine the use of the recovery block pattern in such diverse areas as specification of an embedded control systems, high-level circuit designs and web-services composition architectures. Even when a design component is highly specific to an application domain, the formulation of the design in a reusable form makes it possible to refactor developments and reuse the same design components.

A design component is created to be interpreted by a computer (most likely a development environment) and, therefore, there is no need for a modeller to investigate the internal details of design component. This prevents the possibility of misinterpretation or incorrect adaptation of a design component to a development and makes the use of components almost effortless since a modeller now does no have to learn about the details of the applied design component. The black-box property helps to ensure that components are not modified during use which is

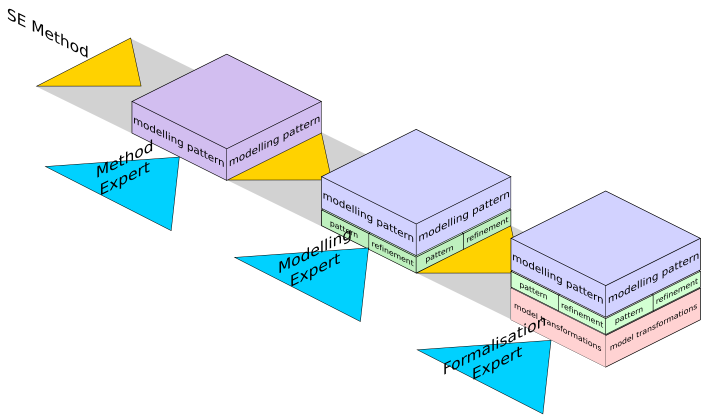


Figure 7.3: Creation of a design component.

important because any change is likely to violate the correctness conditions (Figure 7.1).

In our approach, a design component is made of three layers: the top layer is a collection of modelling patterns, this layer is responsible for high-level guidance during formal development; the midddle layer is a set of refinement or abstraction patterns, they provide automation in the design process; the bottom layer consists of model transformations connecting the two upper layers to a concrete formalism (Figure 7.2).

A simple design component comprises a single modelling pattern and several refinement patterns. We have seen an example of such component in Section 5.3 when we discussed a code generation modelling pattern and then constructed several refinement patterns to support the application of the modelling pattern.

It is possible to have a design component made of a single modelling pattern. We can think of such component as a way to package and distribute the modelling pattern. The same principle works for refinement patterns. One or several refinement patterns can be used to form a design component. From a user's viewpoint, however, such components may be not convenient to work with. A modelling pattern without refinement patterns does not assist a modeller in constructing refinement steps that lead to intermediate modelling goals. A refinement packaged as a design component is simply hard to integrate as a modeller gets no indication at what stage of a development process the pattern should be used. Hence, the adiditonal effort is required to learn the pattern purpose and remember to use it at the right moment.

The construction procedure for a simple refinement design component (the case of an abstraction component is symmetrical) has a number of distinct steps. First, some domain-specific software engineering method is interpreted by a *method ex-*

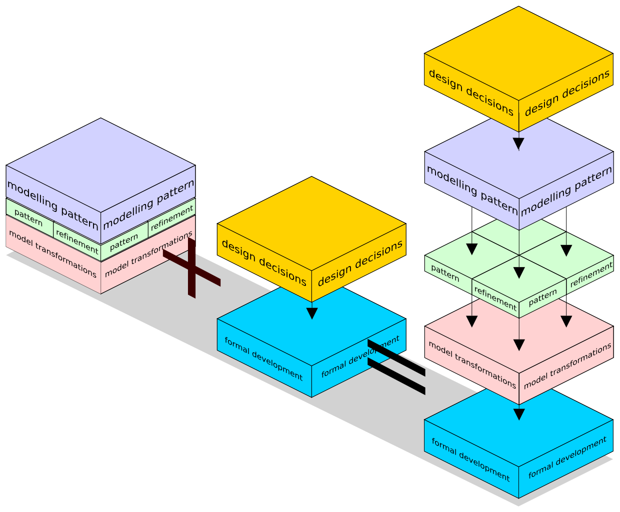


Figure 7.4: A design component assists a modeller by providing refinement automation and modelling tactics.

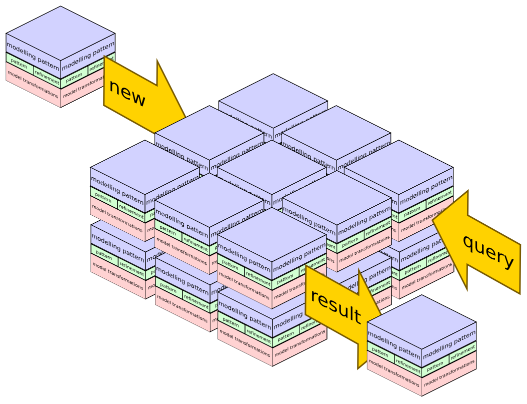


Figure 7.5: Global design component repository may be used to exchange reusable components.

pert to construct a semi-formal abstract modelling pattern. The result is used by a *modelling expert* who selects a suitable formalisation style, further detalises the modelling pattern and, possibly, constructs some method-neutral refinements pattern. Finally, a *formalisation expert* completes the development of the modelling pattern by adding more refinement patterns and demonstrating their correctness in the context of the chosen modelling method (Figure 7.3).

A design component actively participates in a development process. Whereas in traditional, unassisted development, a modeller would translate a mental model directly into a formal model, with design components a modeller manipulates a model with the help of abstraction and refinement patterns and within the limits imposed by a combination of active modelling patterns. A design component takes

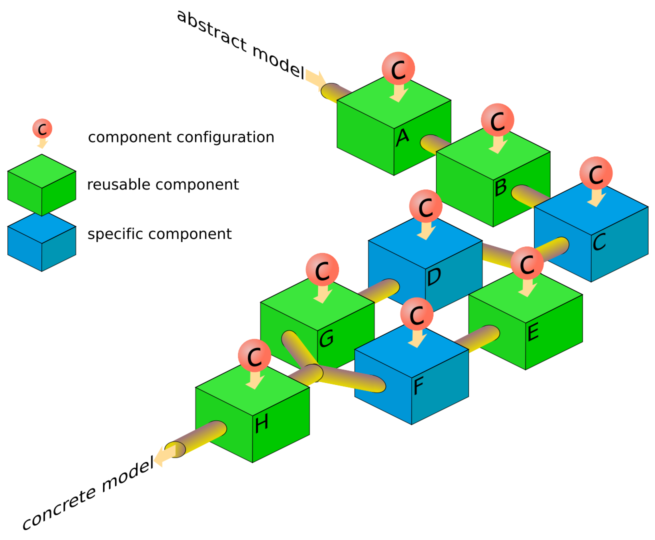


Figure 7.6: A formal development is a pipeline constructed from reusable and development-specific design components.

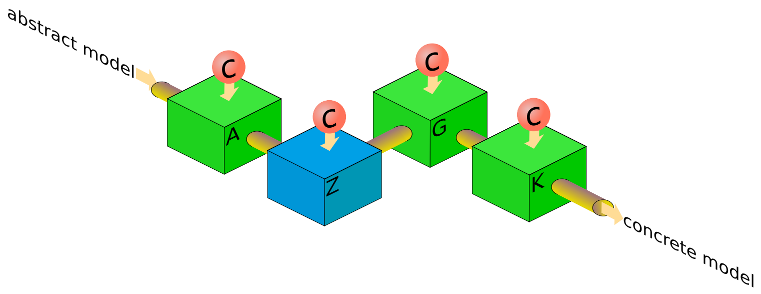


Figure 7.7: Design components can be shared across developments.

care of reusable, generic parts of a development to let a modeller focus on design decisions unique for the modelled system (Figure 7.4).

Reuse is central to software construction. Any software system, large or small, relies on a vast number of third-part reusable components. For anything, but tiniest programs, reuse is an absolute necessity. Design components have all the required properties to facilitate design sharing: they hide internal details but the description of their input and outputs is available. An online component library can be used to automatically discover components providing required designs (Figure 7.4).

With design components, a formal development is realised as a number of components connected together (Figure 7.6). Some of these components are specific to the development and others are imported from a component library. Component based view on development helps to break free from the inflexible refinement chain development while preserving the refinement-based development itself. A pattern-

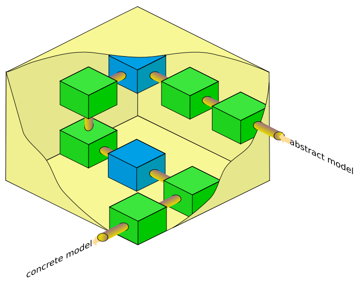


Figure 7.8: A complex design component can be composed from other design components.

based development has the pipeline architecture: there is a single input, where an an abstract model is supplied and one or more outputs which produce modelling results. In addition, each component of a development has a configuration input.

The traditional refinement chain corresponds to the sequential composition of design components, that is, a straight pipeline accepting an input abstract model and producing some concrete model. With our approach, development structuring can be more flexible. Not only we are able to construct a development from reusable bricks - design components - but we can also use parallel composition for cases when different model parts are refined (abstracted) independently.

A component-based development is better structured and thus is easier to refactor. Each component caters for a specific aspect of the development and there is no problem of handling a large, unreadable specification which is often a result of development process.

The same design component can be used in different developments. For example, a new development can be constructed by reusing components A and G from the development on Figure 7.7, adding new third-party component K, and developing a new component Z.

A development constructed from design components can be itself seen as a design component in the context of a larger development (Figure 7.8). Complex components can be constructed using the pipeline architecture from the existing components and a large development can be decomposed into a set of independently developed design components.

7.2 Taxonomy

In the work we have discussed a variety of ways to obtain one model from another. The include

model transformation a simple rule changing model at the syntactic level by adding new elements, updating existing elements and removing existing elements

reversable model transformation an information-preserving model transformation, e.g. a transformation only adding new model elements

inverse model transformation an inverse form of a model transformation that undoes effect of its counterpart; only reversible transformations have an inverse form

pattern a composition of model transformations rules expressed using the pattern language; any pattern must be either an abstraction pattern or a refinement pattern

refinement pattern produces a valid refinement of any accepted input model

abtraction pattern produces a valid abstraction of any accepted input model

superposition pattern a pattern that only adds (removes) elements to construct a refinement (abstraction)

mapping pattern a special case of the superposition pattern that adds only top-level model elements

(de-)intergration pattern a superposition pattern that is applicable to any model (the refinement pattern case) or accept models with a specific elements and removes those elements (the abstraction pattern case)

presentation pattern renames a model element or implements a reversible data refinement

model difference (delta) a pattern constructed mechanically by comparing two models

modelling pattern a set of rules describing an overall model development strategy

abstract modelling pattern a modelling pattern containing informal or undefined predicates in assumptions or goals

concrete modelling pattern a modelling pattern which assumptions and goals can be evaluated to truth or false for any given model

design component a combination of related modelling patterns, refinement patterns, abstraction patterns and model transformation rules

refinement design component a design component that helps to construct series of refinement steps

abstraction design component a design component that helps to reverse-engineer a model

reusable design component a design component made specifically to be reused in different developments

specific design component a component relying on model differences or some development-specific solutions

7.3 Related Works

The pattern reuse mechanism proposed in [48] facilitates reuse of design patterns in formal modelling. The proposal is based on somewhat arguable basis that design patterns can be adequately expressed as B machines and treated as normal B machines. The authors discuss three pattern instantiation methods which are all based on simple conjoining a pattern model with the main development model. A similar approach, although with the focus on object orientation, is described in [49]. We believe that these two approaches are highly inflexible due to the limitations of the employed instantiation mechanisms and the tendency to accumulate many details in a single machine. Besides, we would argue that the B language is inadequate for specifying reusable patterns as it oriented towards different kind of models.

Specification can also be used to facilitate component matching and validation of component composition. Zaremski and Wing [98, 99] present a method for matching software components based on formal specifications of component functionality. Compatibility conditions are stated in the form of theorems and a theorem prover is used to discharge them. This is a bottom-up approach to software constructions. In our proposal we expect a modeller to rely on the top-down refinement procedure and import design rather than implementations.

Abrial's Mechanical Press Controller [88] case study uses patterns to simplify development and reuse proofs. The patterns are constructed independently from the main development using the normal tools and techniques. Pattern instantiation is manual but fairly straightforward: pattern elements are replaced with corresponding model elements. With our approach, this pattern type is covered by the integration pattern class, presented in Section 4.11.3. The case study itself may serve as a benchmark for our proposal and, once the new patterns tool is available, we plan to redevelop the case study using refinement patterns and model differencing. The Automatic Refinement [47] is a technique for producing implementable B model (in B0 language). The approach is based on a collection of model rewrite rules that try to eliminate non-determinism. A large number of such rules were defined, although an individual rule is rather simple. An interesting aspect of the technique is that it works in the context of a specific development method. The method is narrowly oriented on modelling train system and, if followed correctly by a developer, should guarantee that the resulting non-deterministic model can be mechanically translated into an executable implementation. We believe that the model transformations used by the approach can be implemented as refinement patterns. The development method supporting the technique is interesting to us as a possible case study. We believe that it should be possible to construct a number of design components to cover the transformations and the method in an integral manner.

Design patterns, while applicable to a wide range of applications, are not reusable in the sense of component reusability: a developer cannot simply add a design pattern into a program. Meyer and Arnout [83, 100] investigated the possibility of building a conventional software component implementing the idea behind a design pattern. They have found that roughly a half of design pattern can be "componentized" - converted into software components. The idea is that such component would provide all the functionality associated with a design pattern without requiring the effort to implement the pattern. The benefit comes at a cost of reduced flexibility - a developer has to use one particular design pattern implementation - and decreased performance. The spirit of the approach is close to our proposal of design components although these two are applied at different levels of abstraction: we believe it is more fruitful to capture and reuse design ideas and patterns during the modelling stage. It is not surprising the authors could not convert architectural design into components - an abstract rule prescribing an overall system architecture has no place in a programming language. Many design pattern from [39] cannot be handled by refinement patterns as well. However, we are confident that they can be represented in one way or another using modelling patterns.

7.4 Limitations

We have presented a method for formal refinement automation. We believe that the proposal is a step towards establishing formal modelling as a widely-accepted software engineering technique. The mechanism makes formal modelling more accesible to non-experts, it has the potential to make formal developments cheaper and quicker and makes it practical to construct large-scale developments with many refinement steps. A large number of challenging problems have to be addressed before all these goal can be trully achieved.

It is unclear if the critical mass of design components and patterns can ever be accumulated to make affect the practice of formal modelling. Clearly, we cannot hope that design components would emerge from somewhere in large quantities. We can start, however, with a less ambitious goal of collecting refinement and modelling patterns and offering them to developers.

Althougn the concept of refinement automation is very simple, the act of formulating and describing patterns is quite different from what is normally done in formal modelling. From our limited experience with Event-B refinement patterns, creation of a pattern requires deep knowledge of both the formal method applied and the problem domain of the pattern. We have found that it is easier to start with an existing refinement steps and then construct a refinement pattern.

A related disadvantage, is that the use of design components requires new expertise and correspodingly additional training. On the other hand, this can be compensated by the ease of modelling with ready-made components.

Powerfull pattern matching mechanism is absolutely essential to maintain a pattern library. For now, we have to rely on pattern developers to supply accurate pattern annotations and connect them to some global ontology. This may prove an overly optimistic assumption.

The initial ideas of the pattern language and its proof theory were developed during the work on various formalisation projects using the B and Event-B methods. As with any notational system, the pattern mechanism is very likely to be biased towards the B style modellling and state-based formalisms in general. An attempt was made to minimize the notational part to avoid this effect. It is, however, unlikely that the attempt was completely successful. We expect that a different style of pattern language and proof theory will be needed to support different modelling styles. For instances, process algebras represent models as plain expression. The lack of structuring constructs makes it more difficult to reason about model parts so that a completely different model rewriting mechanism is required.

Assembly of a development from design patterns is not a top-down procedure. In principle, a modeller could first constructs a modelling pattern in the top-down manner and then detalise it with third-party design components. This, however, does not work. It is impossible to arrive to a modelling pattern that makes an efficient use of pre-existing design components without knowing what components are available. Therefore, there is a danger that a system design could be skewed to favour design decisions supported by the existing patterns.

In this work we propose using design components to conduct large-scale formalisation projects. We are unable, however, at this moment, to support our claim that design components are fit for the purpose with a success story based on a large-scale case study. We are planning to work on such case study in future within the DEPLOY project [101].

7.5 Future Work

The work described in the thesis can be extended in a number of ways. The first goal is to provide a tool support for both refinement and modelling patterns. We plan to realise such tool as a new version of the RODIN Platform *finer* plugin. The tool will provide a visual enviornment for creation and verification of patterns and a new Eclipse perspective supporting differencing-based Event-B developments.

There is a well-known collection of design patterns [39], and we have proposed two mechanism to capture design decisions: refinement and abstraction patterns, and modelling patterns. It is interesting to explore the mapping between design patterns, refinement patterns and modelling patterns. In particular, it is important to try to discover and address any problems that could undermine simple, faithful and usable representation of known design patterns.

Currently, model transformations, refinement and abstraction patterns, and design components all have single input and output. It is interesting to explore design components accepting more than one input or producing more than one output or both. One possible application of a design component with multiple outputs is a component realising model decomposition. Several models may be conjoined in various ways using a design components with multiple inputs.

The notation used to express refinement and abstraction patterns is essentially a simple declarative programming language. Any part of a pattern is a function producing a new model. Patterns do not use variables and thus there are no side-effects. A possible approach to improve the language expressiviness is to base it on an existing functional language, such as ML [102]. A promising approach is to specify patterns directly in ML to benefit from the connections to theorem provers, like HOL [89, 90].

We realise that our work can be taken further by developing a solid mathematical foundation. Much of the notation used now is ad-hoc and there is little in the sense of a general framework relating model transformation, refinement patterns and modelling patterns. As a future work, we plan to investigate the applicability of the graph transformation approach [103], in particular graph grammars [104], to represenation and analysis of model transformations. We also plan to use the category theory [105] to link together different parts of the proposal, and, possibly, indentify some missing links and new interesting properties.

Bibliography

[1] Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming **8**(3) (1987) 231–274

[2] Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1991)

[3] Emerson, E.A.: Temporal and Modal Logic. In van Leeuwen, J., ed.: Handbook of theoretical computer science (vol. B): formal models and semantics, Cambridge, MA, USA, MIT Press (1990) 995–1072

[4] Hoare, C.A.R.: Communicating Sequential Processes. Commun. ACM **21**(8) (1978) 666–677

[5] Pnueli, A.: The temporal logic of programs. In: FOCS, IEEE (1977) 46–57

[6] Emerson, E.A., Halpern, J.Y.: 'Sometimes' and 'not never' revisited: on branching versus linear time temporal logic. J. ACM **33**(1) (1986) 151–178

[7] Atlee, A.P., Gannon, H.: Specifying and Verifying Requirements of Real-Time Systems. IEEE Transaction on Software Engineering **19**(1) (1993) 41–55

[8] Lamport, L.: The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems **16**(3) (1994) 872–923

[9] Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA (1981)

[10] Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1982)

[11] R.Milner, J., D.Walker: A calculus of mobile processes. Information and Computation, 100 (1992) 1–77

[12] Dijkstra, E.: A Discipline of Programming. Prentice-Hall International (1976)

[13] Abrial, J.R., Schuman, S.A., Meyer, B.: Specification language. In McKeag, R.M., Macnaughten, A.M., eds.: On the Construction of Programs. Cambridge University Press (1980) 343–410

[14] Back, R.J., Sere, K.: Stepwise Refinement of Action Systems. In van de Snepscheut, J.L.A., ed.: Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University, London, UK, Springer-Verlag (1989) 115–138

[15] Hedman, E.J., Kok, J.N., Sere, K.: Coordinating Action Systems. Theor. Comput. Sci. **240**(1) (2000) 91–115

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

[17] Jones, C.B.: Systematic software development using VDM. Prentice Hall International (UK) Ltd. (1986)

[18] Abrial, J.R.: Extending B without changing it (for developing distributed systems). In Habrias, H., ed.: 1st Conference on the B method, IRIN Institut de recherche en informatique de Nantes (1996) 169–190

[19] Abrial, J.R., Mussat, L.: Introducing Dynamic Constraints in B. In: Second International B Conference, LNCS 1393, Springer-Verlag (April 1998) 83–128

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

[21] RODIN Event-B Platform. `http://rodin-b-sharp.sourceforge.net/` (2007)

[22] Escher Technologies: Perfect Developer. `http://www.eschertech.com/` (2008)

[23] Abrial, J.R., Cansell, D.: Click'n'Prove: Interactive Proofs within Set Theory. In Basin, D.A., Wolff, B., eds.: Theorem Proving in Higher Order Logics. Volume 2758 of Lecture Notes in Computer Science., Springer (2003) 1–24

[24] Clearsy: AtelierB: User and Reference Manuals. (2008) Available at `http://www.atelierb.societe.com/indexuk.html`.

[25] Saaltink, M.: The Z/EVES System. In Bowen, J.P., Hinchey, M.G., Till, D., eds.: ZUM'97: Z Formal Specification Notation. Volume 1212 of Lecture Notes in Computer Science., Springer-Verlag (1997) 72–85

[26] Plagge, D., Leuschel, M.: Validating z specifications using the probanimator and model checker. In Davies, J., Gibbons, J., eds.: IFM. Volume 4591 of Lecture Notes in Computer Science., Springer (2007) 480–500

[27] Douglas, J., Kemmerer, R.A.: Aslantest: A Symbolic Eexecution Tool for Testing Aslan Formal Specifications. In: ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM (1994) 15–27

[28] Hekmatpour, S., Ince, D.C.: Software Prototyping, Formal Methods, and VDM. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1988)

[29] Fischer, C., Wehrheim, H.: Model-Checking CSP-OZ Specifications with FDR. In Araki, K., Galloway, A., Taguchi, K., eds.: IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods, London, UK, Springer-Verlag (1999) 315–334

[30] Holzmann, G.J.: The Model Checker SPIN. Software Engineering **23**(5) (1997) 279–295

[31] Khomenko, V.: Model Checking Based on Prefixes of Petri Net Unfoldings. PhD thesis, Newcastle University (2003)

[32] Bernot, G., Gaudel, M.C., Marre, B.: Software Testing Based on Formal Specifications: A Theory and A Tool. Softw. Eng. J. **6**(6) (1991) 387–405

[33] Richardson, D.J., Aha, S.L., O'Malley, T.O.: Specification-based test oracles for reactive systems. In: ICSE '92: Proceedings of the 14th international conference on Software engineering, New York, NY, USA, ACM (1992) 105–118 Chairman-Tony Montgomery.

[34] Morgan, C.: Programming From Specifications. Prentice Hall International (UK) Ltd. (1994)

[35] Back, R.J.J., Wright, J.V.: Refinement Calculus: A Systematic Introduction. Springer-Verlag New York, Inc. (1998)

[36] Srinivas, Y.V., Jullig, R.: Specware: Formal support for composing software. In: Mathematics of Program Construction. (1995) 399–422

[37] Christopher, A.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, USA, 1216. ISBN 0195019199 (1977)

[38] Beck, K., Cunningham, W.: Using Pattern Languages for Object-Oriented Programs. Technical Report No. CR-87-43 (1987)

[39] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley. ISBN 0-201-63361-2 (1995)

[40] Morgan, C.: Programming from specifications. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1994)

[41] Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. Sci. Comput. Program. **9**(3) (1987) 287–306

[42] Snook, C., Butler, M., Oliver, I.: The UML-B Profile for formal systems modelling in UML. In Mermet, J., ed.: UML-B Specification for Proven Embedded Systems Design, Springer (2004)

[43] Snook, C., Butler, M.: U2B - A tool for translating UML-B models into B. In Mermet, J., ed.: UML-B Specification for Proven Embedded Systems Design, Springer (2004)

[44] Ilic, D., Troubitsyna, E., Laibinis, L., Snook, C.: Formalizing UML-based Development of Fault Tolerant Control Systems. In: Workshop on Methods, Models and Tools for Fault Tolerance, Integrted Formal Methods 2007. (2007) 70–79

[45] Pons, C.: Heuristics on the Definition of UML Refinement Patterns. Springer (2006)

[46] Sunye, G., Guennec, A.L., Jquel, J.M.: Design Patterns Application in UML. In Bertino, E., ed.: ECOOP'2000. Volume 1850 of Lecture Notes in Computer Science. (2000) 44–62

[47] Burdy, L., Meynadier, J.M.: Automatic Refinement. Workshop on Applying B in an industrial context : Tools, Lessons and Techniques - Toulouse, Formal Methods 99 (1999)

[48] Blazy, S., Gervais, F., Laleau, R.: Reuse of Specification Patterns with the B Method. In: ZB 2003: Formal Specification and Development in Z and B. Volume abs/cs/0610097. CoRR (2003) 626–626

[49] Chan, E., Robinson, K., Welch, B.: Patterns for B: Bridging Formal and Informal Development. In Julliand, J., Kouchnarenko, O., eds.: B 2007: Formal Specification and Development in B. Volume 4355 of Lecture Notes in Computer Science., Springer (2007) 125–139

[50] Flores, A., Moore, R., Reynoso, L.: A Formal Model of Object-Oriented Design and GoF Design Patterns. In Oliveira, J.N., Zave, P., eds.: FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity, London, UK, Springer-Verlag (2001) 223–241

[51] Eden, A.H., Hirshfeld, Y., Yehudai, A.: LePUS - A Declarative Pattern Specification Language. Technical Report 1998-326, Department of Computer Science, Tel Aviv University (1999)

[52] Eden, A.H., Hirshfeld, Y., Yehudai, A.: Towards a Mathematical Foundation for Design Patterns. Technical Report 1999-004, Department of Computer Science, Tel Aviv University (1999)

[53] Eden, A.H., Yehudai, A., Gil, J.: Precise Specification and Automatic Application of Design Patterns. In: ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE), Washington, DC, USA, IEEE Computer Society (1997) 143–153

[54] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 411–420

[55] Dietrich, J., Elgar, C.: Towards a Web of Patterns. Web Semant. **5**(2) (2007) 108–116

[56] Yahoo: Design Pattern Library. `http://developer.yahoo.com/ypatterns/` (2008)

[57] Grand Challenge 6 : Dependable Systems Evolution. `http://www.fmnet.info/gc6` (2004)

[58] McIlroy, M.D.: Mass-produced Software Components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering) (1976) 88–98

[59] Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)

[60] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough Static Analysis of Device Drivers. SIGOPS Oper. Syst. Rev. **40**(4) (2006) 73–85

[61] Leavens, G.T., Abrial, J.R., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, New York, NY, USA, ACM (2006) 221–236

[62] Randell, B.: System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering. IEEE Press **SE-1**(2) (1975) 220–232

[63] Finer Plugin. `http://finer.iliasov.org` (2008)

[64] Dunne, S.: A Theory of Generalised Substitutions. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B, London, UK, Springer-Verlag (2002) 270–290

[65] Boehm, B.W.: Software Engineering Economics. Prentice-Hall (1981)

[66] Bell, T.E., Thayer, T.A.: Software requirements: Are they really a problem? In: ICSE '76: Proceedings of the 2nd international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1976) 61–68

[67] Letier, E., van Lamsweerde, A.: Deriving operational software specifications from system goals. In: SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, New York, NY, USA, ACM (2002) 119–128

[68] Lamsweerde, A.V., Darimont, R., Massonet, P.: Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In: RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering, Washington, DC, USA, IEEE Computer Society (1995) 194

[69] Darimont, R., van Lamsweerde, A.: Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In Garlan, D., ed.: Foundations of Software Engineering. Volume 21., New York, NY, USA, ACM (1996) 179–190

[70] Yu, E.S.K.: Modeling organizations for information systems requirements engineering. (1993) 34–41

[71] Darimont, R., Delor, E., Massonet, P., Lamsweerde, A.: Grail/kaos: An environment for goal-driven requirements analysis, integration and layout. In: RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97), Washington, DC, USA, IEEE Computer Society (1997) 140

[72] Wirth, N.: Program Development by Stepwise Refinement. Commun. ACM **14**(4) (1971) 221–227

[73] Smith, B.: Logic and Formal Ontology. Husserl's Phenomenology: A Textbook (1989) 29–67

[74] Genesereth, M.R., Fikes, R.E.: Knowledge Interchange Format, Version 3.0 Reference Manual. Technical report, Computer Science Department, Stanford University, Technical Report Logic-92-1 (1992)

[75] Knowledge Sharing Effort public library. `http://www-ksl.stanford.edu/knowledge-sharing/` (1998)

[76] Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Syst. J. **45**(3) (2006) 621–645

[77] Mapelsden, D., Hosking, J., Grundy, J.: Design pattern modelling and instantiation using DPML. In: CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2002) 3–11

[78] OMG: CBOP, DSTC, and IBM. MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-03 (2008)

[79] Iliasov, A.: Refinement Patterns for Rapid Development of Dependable Systems. Proc. Engineering Fault Tolerant Systems Workshop (at ESEC/FSE, Croatia), ACM (September 4, 2007)

[80] Iliasov, A.: Refinement Patterns for Fault Tolerant Systems. Technical Report CS-TR N 1074, School of Computing Science, Newcastle University (2008)

[81] Iliasov, A., Romanovsky, A.: Refinement Patterns for Fault Tolerant Systems. In: EDCC 7: the Seventh European Dependable Computing Conference (EDCC-7). (in press) IEEE CS. (2008)

[82] Iliasov, A.: Refinement Patterns. Technical Report CS-TR N 1075, School of Computing Science, Newcastle University (2008)

[83] Arnout, K.: From Patterns to Components. PhD thesis, Swiss Federal Institute of Technology, Zurich (ETH Zurich) (2004)

[84] Iliasov, A., Romanovsky, A., Arief, B., Laibinis, L., Troubitsyna, E.: 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, Washington, DC, USA, IEEE Computer Society (2007) 141–145

[85] Back, R.J., Sere, K.: Superposition Refinement of Reactive Systems. Formal Aspects of Computing **8**(3) (1996) 324–346

[86] Bauer, F.L., Möller, B., Partsch, H., Pepper, P.: Formal Program Construction by Transformations-Computer-Aided, Intuition-Guided Programming. IEEE Transaction on Software Engineering **15**(2) (1989) 165–180

[87] Mester, A., Krumm, H.: Composition and refinement mapping based construction of distributed applications. In Engberg, U.H., Larsen, K.G., Skou, A., eds.: Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS *(Aarhus, Denmark, 19–20 May, 1995)*. Number NS-95-2 in Notes Series, Department of Computer Science, University of Aarhus, BRICS (1995) 290–303

[88] Abrial, J.R.: Case study of a complete reactive system in Event-B: A Mechanical Press Controller. (2005)

[89] Gordon, M.J.C., Melham, T.F., eds.: Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, New York, NY, USA (1993)

[90] Paulson, L.C.: Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science, 828. Springer, Berlin, Germany (1994)

[91] Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, New York, NY, USA, ACM (2000) 35–46 Chairman-Anthony Finkelstein.

[92] Butler, M.J., Waldén, M.: Distributed System Development in B. In Habrias, H., ed.: First International B Conference, IRIN (Institut de Recherche en Informatique de Nantes) (1996) 155–168

[93] Lemick Web-Page. `http://lemick.sf.net` (2008)

[94] Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-Directed Requirements Acquisition. Science of Computer Programming **20**(1-2) (1993) 3–50

[95] Avizienis, A.: The N-Version Approach to Fault-Tolerant Software. In: IEEE Transaction on Software Engineering SE-11, 12 (December). (1985) 1491–1501

[96] Hamming, R.W.: Coding and information theory. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1986)

[97] Rodin: EU research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems). (2007)

[98] Zaremski, A.M., Wing, J.M.: Signature Matching: A Tool for Using Software Libraries. ACM Transactions on Software Engineering and Methodology **4**(2) (1995) 146–170

[99] Zaremski, A.M., Wing, J.M.: Specification Matching of Software Components. ACM Transactions on Software Engineering and Methodology **6**(4) (1997) 333–369

[100] Meyer, B., Arnout, K.: Componentization: The Visitor Example. Computer **39**(7) (2006) 23–30

[101] Deploy Project Web Site. `http://deploy-project.eu/index.html` (2008)

[102] Milner, R., Tofte, M., Harper, R., Macqueen, D.: The Definition of Standard ML - Revised. The MIT Press (1997)

[103] Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.J., Kuske, S., Plump, D., Schürr, A., Taentzer, G.: Graph Transformation for Specification and Programming. Sci. Comput. Program. **34**(1) (1999) 1–54

[104] Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 3: Concurrency, Parallelism, and Distribution. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999)

[105] Pierce, B.C.: Basic Category Theory for Computer Scientists (Foundations of Computing). The MIT Press (1991)

Appendix A

Event-B Model Transformations

A.1 Model Transformation Scopes

Constructing refinement patterns it is beneficial to cluster pattern rules into groups concerned with different model parts. This makes it possible to analyse pattern correctness by independently analyzing correctness of each group. To facilitate construction of refinement pattern with many independent rule clusters, model transformation descriptions are extended with the **scope** attribute. This attribute defines two sets: model parts on which the transformation depends upon but does not modify (set R) and model parts updated or created by the model transformation (set W). Also

$$R \subset \mathrm{CFL} \qquad W \subset \mathrm{CFL}$$

Where set CFL is constructed from elements of the following two classes

- model elements, identified by name
- an attribute of a model element, identified by an attribute name and model element

The latter is written as $e : \mathtt{att}$. For example, a model transformation rule changing name of an event, would have For $e : \mathrm{nom}$ in W set, where e is the parameter name by which the event is known to the transformation.

Two following two simple rules are used to construct scope of a transformation

- any model element mentioned in requirements or transformation rule makes rule, must be included in the R set;
- a newly constructed model element is added to the W set;

- change of an element attribute results in addition of $e.\mathrm{att}$to the W set, where e is the element and att is the updated attribute.

Two model transformation can be parallel only if their scopes do not intersect. Intersection of transformation scopes is computed as intersection of elements updated by the transformations and pair-wise intersection of elements updated by one transformation and listed as dependencies of another

$$inter((R_1, W_1), (R_2, W_2)) = (W_1 \cap W_2) \cup (R_1 \cap W_2) \cup (R_2 \cap W_1)$$

Set pair (R, W) is defined by a list of elements. Elements prefixed with $\star$ are elements of W, the rest are elements of R.

A.2 Variable Transformations

The following transformation adds a new system variable. The new variable has some arbitrary label, type and initialisation. Addition of a new variable always results in a valid model refinement.

```
addvar($v$)  ≡  requirements
                  $v \in (\mathrm{VAR} \setminus dom(mvar))$
                effect
                  $mvar' = mvar \cup \{v \mapsto (L_v, A_v, T_v)\}$
                scope
                  $\star v, L_v$
                ref
                  $true$
```

The next transformation changes the name of a system variable.

```
varlabel($v, nl$)  ≡  requirements
                        $nl \in FreshVarNames \wedge$
                        $mvar(v) = (l, a, t)$
                      effect
                        $mvar' = mvar \lhd\!\!\!- \{v \mapsto (nl, a, t)\}$
                      scope
                        $v, \star v{:}\mathrm{nom}$
                      ref
                        $false$
```

The transformation below provides an initialisation action for a system variable.

$\mathsf{newvaraction}(v, na) \equiv$ requirements
$na \in dom(mact) \wedge$
$mvar(v) = (l, a, t) \wedge$
$\mathsf{PAT\_FIS\_INI}(na) \wedge$
$\mathsf{PAT\_INV\_INI}(na, J) \wedge$
$isAbstrVar(v) \Rightarrow \mathsf{PAT\_RIN\_INV}([\{na\}], [\{\hat{v}.\mathrm{act}\}], J)$
effect
$mvar' = mvar \lhd\!\!\!- \{v \mapsto (l, na, t)\}$
scope
$v, \star v : \mathrm{act}$
ref
$true$

The following transformation changes the typing predicate of a variable.

$\mathsf{newvartype}(v, nt) \equiv$ requirements
$nt \in \mathrm{TYPE} \wedge$
$mvar(v) = (l, a, t) \wedge$
$\mathsf{PAT\_INV\_INI}(a, J\_vt(v, nt)) \wedge$
$isAbstrVar(v) \Rightarrow$
  $\mathsf{PAT\_RIN\_INV}(na, \hat{v}.\mathrm{act}, J\_vt(v, nt))$
effect
$mvar' = mvar \lhd\!\!\!- \{v \mapsto (l, n, nt)\}$
scope
$v, \star v : \mathrm{typ}$
ref
$true$

A.3 Parameter Transformations

Model transformation addpar creates a new event parameter.

$\mathsf{addpar}(a) \equiv$ requirements
$a \in (\mathrm{PARAM} \setminus dom(marg))$
effect
$marg' = mvar \cup \{v \mapsto (L_p, T_p)\}$
scope
$\star a, L_p$
ref
$true$

The following transformation changes event parameter name.

parlabel(a, nl) $\equiv$ requirements
$nl \in \mathit{FreshVarNames} \wedge marg(v) = (l, t)$
effect
$marg' = mvar \Leftarrow \{a \mapsto (nl, t)\}$
scope
$a, \star a\!:\!\mathrm{nom}$
ref
false

The next transformation changes the type of an event parameter.

newpartype(a, nt) $\equiv$ requirements
$nt \in \mathrm{TYPE} \wedge marg(v) = (l, t)$
effect
$marg' = marg \Leftarrow \{a \mapsto (l, nt)\}$
scope
$a, \star a\!:\!\mathrm{type}$
ref
false

A.4 Event Transformations

The transformation adds a new event with no parameters, empty guard ($true$) and no actions (**skip**). Because of the the non-divergence requirement this transformation does not itself result in a valid refinement. The event guard must be replaced with something that would guarantee the termination of all the new events.

addevent(e, l) $\equiv$ requirements
$e \in (\mathrm{EVENT} \setminus dom(mevt)) \wedge \text{PAT\_NEW\_DIV}$
effect
$mevt' = mevt \cup \{e \mapsto (l, \oslash, \oslash, \oslash, \oslash)\}$
scope
$\star e, l$
ref
false

The next transformation adds an abstract event to the *refines* list of a given concrete event.

$\mathsf{refines}(e, nr) \quad \equiv$ requirements

$e \in dom(mevt) \wedge e = (n, r, p, g, a) \wedge nr \in \mathrm{LABEL} \wedge$
$\exists ee \cdot (ee \in dom(mevt_0) \wedge mevt_0(ee) = (l, nr, pp, gg, aa))$

effect

$mevt' = mevt \cup \{e \mapsto (n, r \cup \{nr\}, p, g, a)\}$

scope

$e, \star e\!:\!\mathrm{ref}$

ref

false

The following model transformation renames an event. Since events are never referenced by name from anywhere in a model, an event can be renamed although the *refines* attribute may be needed to link with abstract model events.

$\mathsf{newevtlabel}(e, nl) \quad \equiv$ requirements

$e \in dom(mevt) \wedge e = (l, p, ga) \wedge$
$nl \in FreshNames$

effect

$mevt' = mevt \mathbin{\lhd\!\!-} \{e \mapsto (nl, p, g, a)\}$

scope

$e, nl, \star e\!:\!\mathrm{nom}$

ref

false

Transformation $addguard$ adds a new guard predicate. The resulting event guard is the conjunction of the old an guard and the newly added part.

$\mathsf{addguard}(e, h) \quad \equiv$ requirements

$e \in dom(mevt) \wedge e = (l, p, ga) \wedge$
$h \in \mathrm{GUARD} \wedge$
$isAbstrEvt(e) \Rightarrow \mathsf{PAT\_REF\_GRD}(\hat{e}.\mathrm{guard}, e.\mathrm{guard} \wedge h)$

effect

$mevt' = mevt \mathbin{\lhd\!\!-} \{e \mapsto (l, p, g \cup \{h\}, a)\}$

scope

$e, e\!:\!\mathrm{grd}$

ref

true

The following transformation adds a parameter to an event descriptor.

$\mathsf{addparam}(e, par) \quad \equiv$ requirements
$e \in dom(mevt) \wedge e = (l, p, ga) \wedge$
$par \in \mathrm{PARAM}$
effect
$mevt' = mevt \lhd\!\!\!- \{e \mapsto (l, p \cup \{par\}, g, a)\}$
scope
$e, e\!:\!\mathrm{par}$
ref
false

The following transformation constructs a new actions and adds it to an event.

$\mathsf{addaction}(e, act) \quad \equiv$ requirements
$e \in dom(mevt) \wedge e = (l, p, ga) \wedge act \in dom(mact) \wedge$
$\mathsf{PAT\_REF\_FIS}([\{act\}]) \wedge$
$isAbstrEvt(e) \Rightarrow \mathsf{PAT\_REF\_INV}([a \cup \{act\}], [\hat{e}.\mathrm{act}]) \wedge$
$\neg isAbstrEvt(e) \Rightarrow \mathsf{PAT\_NEW\_INV}([act])$
effect
$mevt' = mevt \lhd\!\!\!- \{e \mapsto (l, p, g, a \cup \{act\})\}$
scope
$e, \star e\!:\!\mathrm{act}$
ref
true

A.5 Invariant Transformations

The next transformation adds a new model invariant.

$\mathsf{addinv}(i) \quad \equiv$ requirements
$i \in \mathrm{INVAR}$
effect
$inv' = inv \cup \{i\}$
scope
inv
ref
false

A.6 Action Transformations

The transformation below constructs a new action from the given variable, style and expression.

$\mathsf{newaction}(a, v, s, e) \equiv$ requirements
$a \in \text{ACTION} \wedge$
$v \in dom(mvar) \wedge s \in \text{STYLE} \wedge e \in \text{EXPR}$
effect
$mact' = mact \lhd\!\!+ \{a \mapsto (\{v\}, s, e)\}$
scope
$\star a$
ref
$false$

The next transformation adds a new variable to an action.

$\mathsf{newactionvar}(a, nv) \equiv$ requirements
$a \in dom(mact) \wedge a = (v, s, e) \wedge$
$nv \in dom(mvar)$
effect
$mact' = mact \lhd\!\!+ \{a \mapsto (v \cup \{nv\}, s, e)\}$
scope
$a, \star a\!:\!\text{var}$
ref
$false$

The following transformation changes the substitution style of an action.

$\mathsf{newactionsty}(a, ns) \equiv$ requirements
$a \in dom(mact) \wedge a = (v, s, e) \wedge$
$ns \in \text{STYLE}$
effect
$mact' = mact \lhd\!\!+ \{a \mapsto (v, ns, e)\}$
scope
$a, \star a\!:\!\text{sty}$
ref
$false$

The next transformation changes the expression part of an action.

$\mathsf{newactionexp}(a, ne) \equiv$ requirements
$a \in dom(mact) \wedge a = (v, s, e) \wedge$
$ne \in \text{EXPR}$
effect
$mact' = mact \lhd\!\!+ \{a \mapsto (v, s, ne)\}$
scope
$a, \star a\!:\!\text{exp}$
ref
$false$

Appendix B

Recovery Block Pattern Source Code

This Appendix shows a XML representation of the Recovery Block pattern we have defined in Section 4.9, in the form accepted by the pattern tool. A pattern declaration starts with a description sections which list the pattern categories, author, version and the general description. All former three are used to locate and match a pattern. The latter is available to a user when a pattern is selected.

```
<pattern>
<preamble>
    <name>RecoveryBlock</name>
    <author>A. Iliasov</author>
    <version major='0' minor='4' revision='0'/>
    <description>
        Introduces a recovery block
    </description>
    <category>By Project/test</category>
    <category>Fault-Tolerance</category>
    <!--
      This patterns helps to tolerate software or hardware
      failure using N blocks which implementations differ
      in software or hardware or both. Checkpointing is used
      to save state before executing a block so that results
      of unsuccessful block execution can be discarded. A
      block execution is followed by an acceptance test. If
      the test passes then the result of the current is used
      as the final results. Otherwise, state is rolled back and
      another block is activated.

      The pattern takes as input a model with two events. One
    of the events is a specification of the desired behaviour.
    The other event is the connection to some external
    recovery or abortion mechanism. During instantiation , the
    pattern also asks for a number which the number of
    behaviour block blocks.

    The further refinements should diversify designs of
    behaviour blocks and adapt test conditions. A good
    starting point for applying this pattern is a desired
    behaviour event with non−deterministic before−after
    predicates , e.g. a:2..55. The conjunction of all
    before−after predicates is the acceptance test used
    by the pattern.
  −−>
</preamble>

<system id='sys'>
  <select:event id='main'>

    <define:variable
                                    scope='sys'
                                    id='branch'
                                    name='$main$_branch'
                                    type='NAT' init='0'>
      <comment>
          This variables defines the currently active
          block. When goes beyond the blocks number
          indicates a failure.
      </comment>
    <define:variable
                                    scope='sys'
                                    id='stage'
                                    name='$main$_stage'
                                    type='0..2' init='0'>
      <comment>
            Defines process stage:
                checkpoint (0),
                execution (1) or
                test (2)
      </comment>
    <!−− create checkpoint event −−>
    <define:event id='checkpoint' name='$main$_chkp'>
      <comment>
```

```
                Saves the current state so that it can be
                rolled back to later
            </comment>
            <define:guard>$stage$ = 0</define:guard>
            <action id='ac' scope='main'>
                <define:variable scope='sys' id='cpvar'
                      name='$ac.variable.name$_chkp'
                      type='$ac.variable.type$'
                      fullinit='$ac.variable.fullinit$'>
                    <comment>
                        Shadow variable for $ac.variable.name$ used
                        for intermediate results
                    </comment>
                    <define:action scope='checkpoint' variable='$cpvar$'>
                      $ac.variable.name$
                    </define:action>
                </define:variable>
            </action>
            <define:action variable='$stage$'>
                <comment>Let the current block execute</comment>
                1
            </define:action>
        </define:event>
        <index id='i' globalid='index' from='1'
                    hint='Number␣of␣alternative␣branches'>

            <!-- create alternative behaviour events -->
            <define:event id='alt' name='$main$_alt_$i.name$'>
                <comment>
                        Behaviour block $i.name$, it is
                        selected when $branch$ = $i.name-1$
                </comment>
                <define:guard>$stage$ = 1</define:guard>
                <define:guard>$branch$ = $i.name-1$</define:guard>
                <define:guard scope='alt' copyfrom='$main$'/>
                <define:variable scope='alt' copyfrom='$main$'/>
                <action id='ac' scope='main'>
                    <define:action scope='alt'
                                variable='$ac.variable.name$_chkp'
                                style='$ac.style$'>
                      $ac.expression$
                    </define:action>
                </action>
        <define:action  variable='$stage$'>
          <comment>Now go to testing</comment>
          2
        </define:action>
      </define:event>

    </index>

    <!-- The test success event. This event is a
         modified version of the abstract event -->
    <define:guard scope='main'>$stage$ = 2</define:guard>
      <comment scope='main'>
                Test for successful block execution.
                 It compares the block results
                                 with the expected behaviour
      </comment>
      <action id='ac' scope='main'>
        <define:guard id='okgrd'>
          <define:formula scope='okgrd'
                  type='before-after-predicate'
                  variable='$ac.variable.name$_chkp'
                  expression='$ac.expression$'
                  style='$ac.style$'/>
          (dummy)
        </define:guard>
      </action>
      <action id='ac' scope='main'>
        <define:action  variable='$ac.variable.name$'>
          $ac.variable.name$_chkp
        </define:action>
      </action>

    <!-- The test failure event. -->
    <define:event id='test' name='$main$_test_fail'>
      <comment>Block failure test</comment>
      <define:guard>$stage$ = 2</define:guard>
      <define:guard id='tstguard' scope='test'>
        <define:formula scope='tstguard'
                            id='guard' operator='or'>
          <action id='ac' scope='main'>
            <define:formula scope='guard'
                    type='before-after-predicate'
                    variable='$ac.variable.name$_chkp'
```

```
                              expression='$ac.expression$'
                              style='$ac.style$'
                              negation='TRUE'/>
                  </action>
              </define:formula>
              (dummy)  <!-- This is a dummy guard that    will be
                                replaced by a formula. We need
                                 it here to avoid the tool asking a user
                                 to type in a guard expression
                    -->
          </define:guard>
          <define:action variable='$branch$'>
              $branch$ + 1
              <comment>Use different block next time</comment>
          </define:action>
          <define:action variable='$stage$'>
              0
              <comment>
                    Go to checkpoint to discard intermediate
                    results and start over again
              </comment>
          </define:action>
        </define:event>

        <select:event id='fail' predicate='me.name!=main.name'>
          <comment scope='fail'>
                This event is used when all the blocks failed
          </comment>
          <define:guard>$branch$ = $index.upto$</define:guard>
        </select:event>
        </define:variable>
        </define:variable>
    </select:event>
</system>
</pattern>
```

Appendix C

Pattern-generated Event-B Examples

C.1 Abstract Model 1

We use the following abstract model as an input for the N-versioning and Recovery Block patterns. In this model, a memory can be updated with the store operation. The read operation is implicit since it does not affect the model state.

```
SYSTEM memory
VARIABLES
    mem
INVARIANT
    mem ∈ 0 .. 255
INITIALISATION
    mem :∈ 0 .. 255
EVENTS
    store  =  BEGIN
                mem :∈ 0 .. 255
              END
    reset  =  BEGIN
                mem := 0
              END
```

C.2 Recovery Block

This model is constructed by applying the Recovery Block refinement pattern (Section 4.9) to the *memory* model. The instantiation parameters are the transformed event $e = store$ and the number of alternatives $n = 3$. This refinement has 18 proof obligations (all automatically discharged by the RODIN platform prover). Note that the correctness proof we have done for the pattern make the proof obligations redundant.

```
SYSTEM $memory\_rb$
REFINES $memory$
VARIABLES
    $mem, store\_branch, store\_stage, mem\_chkp$
INVARIANT
    $mem \in 0 \mathrel{..} 255$
    $store\_branch \in \mathbb{N}$
    $store\_stage \in 0 \mathrel{..} 2$
    $mem\_chkp \in 0 \mathrel{..} 255$
INITIALISATION
    $mem :\in 0 \mathrel{..} 255$
    $store\_branch := 0$
    $store\_stage := 0$
    $mem\_chkp :\in 0 \mathrel{..} 255$
EVENTS
    $store\_chkp$   =   WHEN
                          $(store\_stage = 0)$
                        THEN
                          $mem\_chkp := mem$
                          $store\_stage := 1$
                        END
    $store\_alt\_1$  =   WHEN
                          $(store\_stage = 1) \wedge (store\_branch = 0)$
                        THEN
                          $mem\_chkp :\in 0 \mathrel{..} 255$
                          $store\_stage := 2$
                        END
```

```
$store\_alt\_2$      =  WHEN
                          $(store\_stage = 1) \wedge (store\_branch = 1)$
                        THEN
                          $mem\_chkp :\in 0\,..\,255$
                          $store\_stage := 2$
                        END
$store\_alt\_3$      =  WHEN
                          $(store\_stage = 1) \wedge (store\_branch = 2)$
                        THEN
                          $mem\_chkp :\in 0\,..\,255$
                          $store\_stage := 2$
                        END
$store\_test\_fail$  =  WHEN
                          $(store\_stage = 2) \wedge (mem\_chkp \notin 0\,..\,255)$
                        THEN
                          $store\_branch := store\_branch + 1$
                          $store\_stage := 0$
                        END
$store$              =  WHEN
                          $(store\_stage = 2)$
                        THEN
                          $mem := mem\_chkp$
                        END
$reset$              =  WHEN
                          $(store\_branch = 3)$
                        THEN
                          $mem := 0$
                        END
```

C.3 N-version Programming

The example demonstrates a sample output for the N-versioning refinement pattern (Section 6.2). This model is constructed by applying the pattern to the *memory* model. The instantiation parameters are the transformed event $e = store$ and the number of alternatives $n = 3$. The model has 24 proof obligations, all discharged automatically.

```
SYSTEM $memory\_nvp$
REFINES $memory$
VARIABLES
    $mem, store\_stage, ready\_1, mem\_1,$
    $ready\_2, mem\_2, ready\_3, mem\_3,$
    $store\_result, store\_failure$
INVARIANT
    $mem \in 0 \,..\, 255 \wedge store\_stage \in 0 \,..\, 2 \wedge$
    $ready\_1 \in BOOL \wedge mem\_1 \in 0 \,..\, 255 \wedge$
    $ready\_2 \in BOOL \wedge mem\_2 \in 0 \,..\, 255 \wedge$
    $ready\_3 \in BOOL \wedge mem\_3 \in 0 \,..\, 255 \wedge$
    $store\_result \in \mathbb{N} \leftrightarrow ((0 \,..\, 255)) \wedge store\_failure \in BOOL$
INITIALISATION
    $mem :\in 0 \,..\, 255 \parallel store\_stage := 0$
    $ready\_1 := FALSE \parallel mem\_1 :\in 0 \,..\, 255$
    $ready\_2 := FALSE \parallel mem\_2 :\in 0 \,..\, 255$
    $ready\_3 := FALSE \parallel mem\_3 :\in 0 \,..\, 255$
    $store\_result := \{\} \parallel store\_failure := FALSE$
EVENTS
    $reset$  =  BEGIN $mem := 0$ END
    $alt\_1$  =  WHEN
                  $store\_stage = 0 \wedge ready\_1 = FALSE$
                THEN
                  $mem\_1 :\in 0 \,..\, 255$
                  $ready\_1 := TRUE$
                END
    $alt\_2$  =  WHEN
                  $store\_stage = 0 \wedge ready\_2 = FALSE$
                THEN
                  $mem\_2 :\in 0 \,..\, 255$
                  $ready\_2 := TRUE$
                END
```

```
alt_3          =  WHEN
                    $store\_stage = 0 \wedge ready\_3 = FALSE$
                  THEN
                    $mem\_3 :\in 0\,..\,255$
                    $ready\_3 := TRUE$
                  END
store_collect  =  WHEN
                    $store\_stage = 0)\wedge$
                    $ready\_1 = TRUE \wedge ready\_2 = TRUE\wedge$
                    $ready\_3 = TRUE$
                  THEN
                    $store\_result :=(\{1 \mapsto (mem\_1)\})\cup$
                                    $(\{2 \mapsto (mem\_2)\})\cup$
                                    $(\{3 \mapsto (mem\_3)\})$
                    $store\_stage := 1$
                  END
store          =  ANY $store\_k, mem\_t$ WHERE
                    $store\_stage = 1\wedge$
                    $(\forall j\cdot(j \in dom(store\_result) \wedge j \neq store\_k\Rightarrow$
                        $card(store\_result^{-1}[\{store\_result(store\_k)\}]) \geq$
                            $card(store\_result^{-1}[\{store\_result(j)\}])))\wedge$
                    $((mem\_t) = store\_result(store\_k))\wedge$
                    $store\_k \in dom(store\_result)\wedge$
                    $mem\_t \in 0\,..\,255$
                  THEN
                    $mem := mem\_t$
                    $store\_stage := 2$
                    $ready\_1 := FALSE$
                    $ready\_2 := FALSE$
                    $ready\_3 := FALSE$
                    $store\_failure := bool(card(store\_result^{-1}[$
                                                    $\{store\_result(store\_k)\}]) < 2)$
                  END
```

C.4 Abstract Model 2

We use the following abstract model as an example for the Parity and Hamming refinement patterns. The model specifies a communication protocol based on a shared buffer variable. The buffer ch is only large enough to accomodate a single value. Flag rd is used to indicate the buffer state (empty ($rd = FALSE$) or full ($rd = TRUE$)). The $send$ event generates and writes data into the buffer. The sender copies the values from the buffer into the $outv$ variable.

```
SYSTEM $SendRecv$
VARIABLES
    $ch, outv, rd$
INVARIANT
    $ch \in 0 .. 15$
    $outv \in 0 .. 15$
    $rd \in \mathbb{B}$
    $rd = false \Rightarrow outv = ch$
INITIALISATION
    $ch := 0$
    $outv := 0$
    $rd := false$
EVENTS
    $send$       =  ANY $v$ WHERE
                      $rd = false \wedge v \in 0 .. 15$
                    THEN
                      $ch := v$
                      $rd := true$
                    END

    $receive$    =  WHEN
                      $rd = true$
                    THEN
                      $outv := ch$
                      $rd := false$
                    END
```

C.5 Parity Check Bit

The appendix demonstrates the use of the Parity refinement pattern. The model below is created by applying the Parity pattern (see Section 6.3) to the $SendRecv$ model (Section 6.3.1). The instantiation parameters are: ch (the channel variable); $send$ (the event writing into the channel); $receive$ (the receiver event).

```
SYSTEM $parity$
REFINES $SendRecv$
VARIABLES
    $ch, outv, rd, ch\_ph, ch\_err, ch\_bits$
INVARIANT
    $ch \in 0\,..\,5 \wedge outv \in 0\,..\,5 \wedge$
    $rd \in BOOL \wedge ch\_ph \in 0\,..\,4 \wedge$
    $ch\_err \in BOOL \wedge ch\_bits \in 1\,..\,4 \rightarrow \{0,1\} \wedge$
    $rd = FALSE \Rightarrow outv = ch \wedge$
    $ch\_ph = 2 \Rightarrow (ch = (0 + 1 * ch\_bits(1) + 2 * ch\_bits(2) + 4 * ch\_bits(3))) \wedge$
    $ch\_ph = 4 \wedge ch\_err = FALSE \Rightarrow$
                $(ch = (0 + 1 * ch\_bits(1) + 2 * ch\_bits(2) + 4 * ch\_bits(3)))$
INITIALISATION
    $ch := 0 \parallel outv := 0$
    $rd := FALSE \parallel ch\_ph := 0$
    $ch\_err := FALSE \parallel ch\_bits := 1\,..\,4 \times \{0\}$
EVENTS
    $receive$   =  WHEN
                     $rd = TRUE \wedge ch\_ph = 4 \wedge$
                     $ch\_err = FALSE$
                   THEN
                     $outv := ch$
                     $rd := FALSE$
                     $ch\_ph := 0$
                   END
    $ch\_error$  =  WHEN
                     $rd = TRUE \wedge ch\_ph = 4 \wedge$
                     $ch\_err = TRUE$
                   THEN
                     $outv := ch$
                     $rd := FALSE$
                     $ch\_ph := 0$
                   END
```

```
$send$        =  ANY $v$ WHERE
                   $rd = FALSE \wedge$
                   $ch\_ph = 0 \wedge$
                   $v \in 0\,..\,5$
                 THEN
                   $ch := v$
                   $rd := TRUE$
                   $ch\_ph := 1$
                 END
$ch\_code$    =  ANY $p, bit1, bit2, bit3$ WHERE
                   $ch\_ph = 1 \wedge$
                   $ch = 1 * bit1 + 2 * bit2 + 4 * bit3 \wedge$
                   $p = (bit1 + bit2 + bit3) mod 2 \wedge$
                   $p \in \{0,1\} \wedge bit1 \in \{0,1\} \wedge bit2 \in \{0,1\} \wedge bit3 \in \{0,1\}$
                 THEN
                   $ch\_bits := ch\_bits \Leftarrow (\{1 \mapsto bit1\} \cup \{2 \mapsto bit2\} \cup$
                                      $\{3 \mapsto bit3\} \cup \{4 \mapsto p\})$
                   $ch\_ph := 2$
                 END
$ch\_noise$   =  ANY $bit, val$ WHERE
                   $ch\_ph = 2 \wedge$
                   $bit \in 1\,..\,4 \wedge val \in \{0,1\}$
                 THEN
                   $ch\_bits := ch\_bits \Leftarrow \{bit \mapsto val\}$
                   $ch\_ph := 3$
                 END
$ch\_decode$  =  ANY $par$ WHERE
                   $ch\_ph = 3 \wedge$
                   $par = (ch\_bits(1) + ch\_bits(2) + ch\_bits(3) + ch\_bits(4)) mod 2) \wedge$
                   $par \in \{0,1\}$
                 THEN
                   $ch\_err := bool(par \neq 0)$
                   $ch\_ph := 4$
                 END
```

The model has 37 proof obligations of which 21 are discharged automatically and the remaining 16 are handled with the interactive prover.

C.6 Hamming Code

This is an example of a refinement step constructed using the Hamming pattern (see Section 6.3). The model is created by applying the Hamming pattern to the $SendRecv$ model (Section 6.3.1). The instantiation parameters are: ch (the channel variable); $send$ (the event writing into the channel); $receive$ (the receiver event).

```
SYSTEM $hamming$
REFINES $SendRecv$
VARIABLES
    $ch, outv, rd, ch\_ph, ch\_err, ch\_bits$
INVARIANT
    $ch \in 0 \,..\, 5 \land outv \in 0 \,..\, 5 \land$
    $rd \in BOOL \land ch\_ph \in 0 \,..\, 4 \land$
    $ch\_err \in BOOL \land ch\_bits \in 1 \,..\, 5 \rightarrow \{0, 1\} \land$
    $rd = FALSE \Rightarrow outv = ch \land ch\_ph = 2 \Rightarrow$
                    $(ch = (0 + 1 * ch\_bits(3) + 2 * ch\_bits(5))) \land$
    $ch\_ph = 4 \land ch\_err = FALSE \Rightarrow$
                    $(ch = (0 + 1 * ch\_bits(3) + 2 * ch\_bits(5))) \land$
    $ch\_err = FALSE$
INITIALISATION
    $ch := 0 \parallel outv := 0$
    $rd := FALSE \parallel ch\_ph := 0$
    $ch\_err := FALSE \parallel ch\_bits := 1 \,..\, 5 \times \{0\}$
EVENTS
    $receive$  =  WHEN
                    $rd = TRUE \land$
                    $ch\_ph = 4 \land$
                    $ch\_err = FALSE$
                  THEN
                    $outv := ch$
                    $rd := FALSE$
                    $ch\_ph := 0$
                  END
```

```
$ch\_error$   =   WHEN
                    $rd = TRUE \wedge$
                    $ch\_ph = 4 \wedge$
                    $ch\_err = TRUE$
                  THEN
                    $outv := ch$
                    $rd := FALSE$
                    $ch\_ph := 0$
                  END
$send$        =   ANY $v$ WHERE
                    $rd = FALSE \wedge$
                    $ch\_ph = 0 \wedge$
                    $v \in 0 \mathinner{..} 5$
                  THEN
                    $ch := v$
                    $rd := TRUE$
                    $ch\_ph := 1$
                  END
$ch\_code$    =   ANY $bit3, bit5$ WHERE
                    $ch\_ph = 1) \wedge$
                    $ch = 1 * bit3 + 2 * bit5 \wedge$
                    $bit3 \in \{0, 1\} \wedge bit5 \in \{0, 1\}$
                  THEN
                    $ch\_bits := \{3 \mapsto bit3\} \cup$
                               $\{5 \mapsto bit5\} \cup$
                               $(\{1 \mapsto ((bit3 + bit5) mod 2)\}) \cup$
                               $(\{2 \mapsto (bit3 mod 2)\}) \cup$
                               $(\{4 \mapsto (bit5 mod 2)\})$
                    $ch\_ph := 2$
                  END
```

```
ch_noise = ANY $bit, val$ WHERE
             $ch\_ph = 2 \wedge$
             $bit \in 1\,..\,5 \wedge$
             $val \in \{0, 1\}$
           THEN
             $ch\_bits := ch\_bits \lhd\!\!\!- \{bit \mapsto val\}$
             $ch\_ph := 3$
           END
ch_decode = ANY $par1, par2, par3$ WHERE
             $(ch\_ph = 3) \wedge$
             $(par1 = ((ch\_bits(1) + ch\_bits(3) + ch\_bits(5)) mod 2)) \wedge$
             $(par2 = ((ch\_bits(2) + ch\_bits(3)) mod 2)) \wedge$
             $(par3 = ((ch\_bits(4) + ch\_bits(5)) mod 2)) \wedge$
             $(par1 \neq 0 \vee par2 \neq 0) \wedge$
             $par1 \in \{0, 1\} \wedge par2 \in \{0, 1\} \wedge par3 \in \{0, 1\}$
           THEN
             $ch\_bits := ch\_bits \lhd\!\!\!- (\{$
                           $1 * par1 + 2 * par2$
                             $\mapsto$
                               $(((ch\_bits(1 * par1 + 2 * par2)) + 1) mod 2)\})$
             $ch\_ph := 4$
           END

ch_nodecode = ANY $par1, par2, par3$ WHERE
             $(ch\_ph = 3) \wedge$
             $(par1 = ((ch\_bits(1) + ch\_bits(3) + ch\_bits(5)) mod 2)) \wedge$
             $(par2 = ((ch\_bits(2) + ch\_bits(3)) mod 2)) \wedge$
             $(par3 = ((ch\_bits(4) + ch\_bits(5)) mod 2)) \wedge$
             $(par1 = 0 \wedge par2 = 0) \wedge$
             $par1 \in \{0, 1\} \wedge par2 \in \{0, 1\} \wedge par3 \in \{0, 1\}$
           THEN
             $ch\_ph := 4$
           END
```

The model has 48 proof obligations of which 22 are discharged automatically and 20 more are easily discharged with the interactive prover. The remaining 2 theorems express the essential properties of the Hamming codewords.