UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Towards a Practically Extensible Event-B Methodology

by

Issam Maamria

Thesis for the degree of Doctor of Philosophy

January 2013

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

TOWARDS A PRACTICALLY EXTENSIBLE EVENT-B METHODOLOGY

by Issam Maamria

Formal modelling is increasingly recognised as an important step in the development of reliable computer software. Mathematics provide a solid theoretical foundation upon which it is possible to specify and implement complex software systems. Event-B is a formalism that uses typed set theory to model and reason about complex systems. Event-B and its associated toolset, Rodin, provide a methodology that can be incorporated into the development process of software and hardware. Refinement and mathematical proof are key features of Event-B that can be exploited to rigorously specify and reason about a variety of systems.

Successful and usable formal methodologies must possess certain attributes in order to appeal to end-users. Expressiveness and extensibility, among other qualities, are of major importance. In this thesis, we present techniques that enhance the extensibility of: (1) the mathematical language of Event-B in order to enhance expressiveness of the formalism, and (2) the proving infrastructure of the Rodin platform in order to cope with an extensible mathematical language.

This thesis makes important contributions towards a more extensible Event-B methodology. Firstly, we show how the mathematical language of Event-B can be made extensible in a way that does not hinder the consistency of the underlying formalism. Secondly, we describe an approach whereby the prover used for reasoning can be augmented with proof rules without compromising the soundness of the framework. The theory component is the placeholder for mathematical and proof extensions. The theoretical contribution of this thesis is the study of rewriting in the presence of partiality. Finally, from a practical viewpoint, proof obligations are used to ensure soundness of user-contributed extensions.

Contents

List of Figures

Listings

List of Tables

List of Definitions

Declaration of Authorship

I, Issam Maamria , declare that the thesis entitled *Towards a Practically Extensible Event-B Methodology* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- Jean-Raymond Abrial contributed the theories concerning relations, fixpoint and closure in Chapter 6.
- parts of this work have been published as: [77], [78] and [46]

Signed:.....

Acknowledgements

This work would not have been possible without the support and input from a number of people. Firstly, I would like to thank my supervisor Prof. Michael Butler for his remarkable patience and endless support throughout the four eventful years of my PhD. Making a move to Event-B research was my best decision yet. I would also like to thank Jean-Raymond Abrial, Andy Edmunds, Abdolbaghi Rezazadeh, Laurent Voisin, Matthias Schmalz and Bernd Fischer for their invaluable and insightful input.

The following people have greatly contributed to making this long journey bearable: Dr. Hamza Rouabah, Mourad Khelifa, Dr. Zak Mihoubi, Abdeldjalil Belouettar, Issam Souilah, Bilel Remmache, Dr. Aissa Melouki, Ahmed Nouacer, Dr. Tarek Nechma and Dr. Ahmed Maache.

This PhD would not have been possible without the support of my beloved parents, Hafnaoui and Khadidja, and my very dear uncle, Lezhari. To all these people, I cannot thank you enough for your support over the course of my PhD.

Chapter 1

Introduction

Formal methods refer to the mathematical techniques employed for the specification, development and verification of software and hardware systems. Formal methods can be classified, on the basis of the used methodology, into two broad categories: *verification* methods and *correct by construction* methods. In the verification-based approach, correctness is established post-facto, after the program in question is developed. In the correct by construction approach, on the contrary, the development of the system is carried out in an incremental fashion where each intermediate step is verified. In the latter approach, the formal specification of the system provides the blueprint against which its implementation is evaluated.

Event-B [11, 14] is a mathematical technique that can be incorporated into the development process of hardware and software systems [9]. Event-B can be used to model discrete systems and falls into the 'correct by construction' category. The formalism is based on the B method [8], a method that already has good industrial strength [20]. Event-B modelling is carried out by means of two components (also called constructs): contexts and machines. Contexts define the static aspects of a model; they may include carrier sets and constants, as well as axioms and theorems describing the sets and constants. Machines, on the other hand, describe the dynamics of a model; this includes variables and invariants, as well as events (transitions). Event-B uses set theory built around first-order logic as a vehicle for modelling. Proof obligations are generated from models to verify their consistency with respect to some behavioural semantics [61].

The Rodin platform [31] provides a toolset to carry out specification, refinement and proof in Event-B. Rodin proposes a reactive modelling environment that makes it easier for the user to link models, proof obligations and their corresponding proofs. Since proofs are important to the modelling activity, Rodin provides a proof infrastructure that is extensible. External provers (e.g., Atelier-B provers [7]) can also be used in conjunction with the Rodin internal prover. In this thesis, we explain our approach in dealing with issues related to prover extensibility in the context of Event-B.

1.1 Motivation

The Rodin platform provides a reactive modelling environment where the modeller is constantly informed about the effects of changes made to models. To achieve this objective, the tools that constitute Rodin need to work in a reactive manner [80]. This means that, when a model is modified:

1. It is automatically checked for syntax and type errors.
2. Proof obligations are generated.
3. The status of its proofs are updated.

The proving activity is pivotal to modelling. The modeller may gain considerable insight into his models by inspecting failed proofs; this may guide the modeller to modify the model in such a way that proofs become easier to conduct. In some instances, however, failed proofs may be attributed to limitations in the proving infrastructure, e.g., the absence of certain proof rules.

Despite being optimised for proof reuse [82], the current Rodin architecture[1] has the following limitations:

- in order to add a new proof rule, it is required to implement a rule schema in Java. Therefore, a certain level of competence with the Java programming language as well as knowledge of Rodin architecture are necessary;
- after a new rule is added, soundness of the prover augmented with the new rule has to be established. Although Java verification tools, e.g., JML, can be useful for this purpose, such validation has not been performed for any of the built-in rules[2].

The external provers integrated into the Rodin proving infrastructure, ML and PP [7], do not provide sufficient information about how the proof of a sequent has been achieved. Information such as the set of required hypotheses is important for proof *reuse* and *replay* [82]. These properties of proofs are crucial for the efficient running of a reactive modelling environment.

As well as prover extensibility, we aim to address issues related to language extensibility. The mathematical language of Event-B is based on set theory as constructed in [11]. The abstract syntax tree representing formulae in Event-B cannot be extended with new syntax (adding a new operator for instance). This presents a major issue that hinders

[1] The problem no longer exists in the current platform (v2.6) if the Theory plug-in is installed.
[2] As of July 29th, 2012.

reusability in Event-B. Finally, language and prover extensibility are intrinsically linked as the ability to effectively reason about language extensions is of paramount importance. The following two examples provide concrete justifications for our motivation.

1.1.1 Motivation for Proof Extensions

The following formula is a valid polymorphic Event-B theorem:

$$\forall a, b, f \cdot (a \in \mathbb{P}(A) \land b \in \mathbb{P}(B) \land f \in a \pfun b) \Rightarrow (finite(a) \Rightarrow finite(f)) \ . \tag{1.1}$$

Theorem (1.1) states that a partial function with a finite domain is also finite. Note that A and B are type parameters, and as such the valid theorem is polymorphic on both A and B.

Theorem (1.1) can be written as an inference rule as follows:

$$\frac{a \in \mathbb{P}(A), b \in \mathbb{P}(B), f \in a \pfun b \ \vdash \ finite(a)}{a \in \mathbb{P}(A), b \in \mathbb{P}(B), f \in a \pfun b \ \vdash \ finite(f)} \tag{1.2}$$

which states that to prove that a partial function is finite it is sufficient to prove that its domain is a finite set. At the time of writing this thesis, the above rule was not available as part of Rodin proof infrastructure. In order to add the rule to Rodin, it is required to implement (in Java) a schema rule that incorporates pattern matching. Soundness becomes a concern as soon as new rules are added. Furthermore, this process of specifying new rules presents a challenge for end-users.

In this thesis, we show how it is possible to address both issues of extensibility and soundness in an effective fashion.

1.1.2 Motivation for Mathematical Extensions

Sequences are ordered collection of objects, and can be modelled as functions with finite integer contiguous domains. Sequences are part of the classical B [8] repertoire of mathematical operators. In Event-B, however, the sequence operator is not available. There are ways to overcome such limitation, by overloading the functionality of contexts to define sequences axiomatically (see [99]).

Assuming the availability of a context with a carrier set A and a constant a such that $a \subseteq A$, sequences can be defined as belonging to the set:

$$\{f, n . f \in 1..n \tfun a \mid f\} \ .$$

The issue with the aforementioned definition is that sequences can only be used with sets whose type depend on the carrier set. This is problematic, since in a model, one

might want to use sequences of different types according to the modeller's needs. In this thesis, we show how such operators can be defined in a polymorphic manner to overcome the previous limitation.

1.2 Objectives

Our aim is to improve the overall extensibility of Event-B to enhance usability and effectiveness of the methodology. We are primarily concerned with facilitating the addition of new operators (i.e., *language extensions*) and new proof rules (i.e., *prover extensions*) to suit end-users needs. It is essential to ensure that any technique that achieves the aforementioned targets has to maintain *practicality of use* and ensure *soundness preservation*. Practicality of use is important to relieve end-users from writing Java code. Soundness preservation ensures that any extensions do not compromise the logical foundations of the formalism. The logic of Event-B is extensively studied in [102] where a clear definition of soundness is presented, and our work will build on that. A summary of Schmalz's work [102] including the soundness of Event-B proof calculus is presented in Chapter 2. The following key points summarise the objectives of this work:

1. Provide a mechanism by which users can define operators and datatypes in a familiar fashion (i.e., in line with existing practices of developing models in Rodin) thereby allowing language extensions. The new mechanism needs to adhere to the aforementioned requirements: practicality of use and soundness preservation.

2. Provide a mechanism by which the Rodin proving infrastructure can be augmented with new proof rules. Any newly added rules will have to be validated so that the soundness of the existing prover is not compromised. Rewrite and inference rules are used in Rodin to discharge proof obligations. The following milestones are important in order to achieve this objective:

   (a) provide a unifying study of term rewriting and well-definedness. This is of major importance since the Event-B logic deals with partial functions which may give rise to potentially ill-defined terms. To illustrate the importance of this particular contribution, we consider the following rewrite rule:

$$\begin{array}{rcll} f \lhd \{x \mapsto y\}\ (z) & \rightarrow & x = z & : y \\ & & x \neq z & : f(z) \end{array}$$

   Consider the following expression:

$$\{1 \mapsto 2, 1 \mapsto 3, 2 \mapsto 4\} \lhd \{1 \mapsto 5\}(a)$$

where a is an integer. In the case where $a \neq 1$, the rewritten expression is

$$\{1 \mapsto 2, 1 \mapsto 3, 2 \mapsto 4\}(a)$$

which, in the logic of Event-B, is ill-defined since $\{1 \mapsto 2, 1 \mapsto 3, 2 \mapsto 4\}$ is not a function. The previous rewrite rule has been implemented in Rodin, but was later found unsound as it does not satisfy the conditions singled out in our study.

(b) study how term rewriting can be integrated as a proof step within the well-definedness preserving sequent calculus [82, 81]. Mehta [81] presents a calculus for reasoning in the presence of partial functions. The calculus includes a set of well-definedness preserving inference rules that can be used for deduction in Event-B proofs. Mehta's work was the backbone of the proof infrastructure in Rodin. Our work builds on [82, 81], and considers the addition of rewriting steps to the well-definedness preserving calculus. In particular, we study how conditional rewrite rules can be used alongside the well-definedness preserving inference rules in order to enhance the proving capabilities of Rodin.

3. Show how tool support is provided to achieve the first two objectives. We present the Theory plug-in which addresses the extensibility issues of Event-B as discussed in §1.1. We also show by means of several small case studies how our approach can be incorporated into the modelling and proof activity using Event-B.

1.3 Scope of this Thesis

The work described in this thesis unifies three important fields:

- *logic*: Event-B uses a logic based on set theory which provides facilities for defining and reasoning about partial functions. Suitably, reasoning in Event-B is carried out using a sequent logic that accounts for potentially ill-defined terms [82, 81]. In this thesis, we study how rewriting can be integrated as a proof step within the proof system of Event-B.

- *formal methods*: Usability and extensibility are important attributes of successful formalisms. In this work, we explain our approach to deal with extensibility and the resulting usability issues in the context of Event-B. Prover and language extensibility are important in terms of giving more power to the modeller. However, it is also important to improve the usability of the formalism whilst maintaining soundness and integrity.

- *software engineering*: The ideas presented in this thesis have been used to improve the Event-B toolset. The enhancements made to Rodin allow modellers to define and reason about mathematical extensions in a familiar manner. Mathematical and prover extensions can be readily used in modelling once they are inspected and checked for soundness. The effort required to switch between modelling and meta-reasoning is minimised, since familiar techniques are used to enhance usability and extensibility.

1.4 Publications

1. Issam Maamria and Michael Butler. Rewriting and Well-Definedness within a Proof System. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers PAR'10*, volume 43 of *EPTCS*, pages 49-64, 2010 [77].

2. Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh. On an Extensible Rule-Based Prover for Event-B. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Rgine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 407-407. Springer Berlin / Heidelberg, 2010 [78].

1.5 Outline

This thesis makes important contributions to Event-B in general as described in §1.2. Chapter 2 provides useful background to the reader. It puts this work in context by concisely presenting the different concepts needed for the remainder of the thesis. Event-B and the Rodin toolset are introduced and the limitations of the existing framework are outlined. Moreover, the proof calculus used in Event-B is presented together with a detailed overview of well-definedness and partial functions. The remaining chapters are categorised as follows:

1. **Chapter 3: Rewriting and Well-Definedness within a Proof System** This chapter presents a contribution of a more theoretical nature. A unifying treatment of rewriting and well-definedness is presented to provide the theoretical foundation for the subsequent chapters. Finally, we describe how rewriting can be added as a proof step within the sequent calculus used by Event-B.

2. **Chapter 4: A Practical Approach to Event-B Prover and Language Extensibility** In this chapter, we present the theory component which will be used as a vehicle for defining extensions in Event-B. A detailed description of the approach employed to deal with prover extensibility is presented. We also

show how the theory component can be used to define new polymorphic operators and datatypes. Proof obligations that ensure soundness of extensions are discussed. The Rule-based Prover and the Theory component provide a practical yet soundness-preserving mechanism to address language and prover extensibility issues in the Event-B methodology.

3. **Chapter 5: Tool Support: Theory Plug-in** In this chapter, we introduce the Theory plug-in which embodies the different ideas presented in this thesis.

4. **Chapter 6: Theory Development: Examples** In this chapter, we present several case studies that demonstrate the usefulness of the Theory plug-in.

Chapter 7 concludes the thesis and summarises its main contributions. Possible areas for future work are outlined.

Chapter 2

Background

In this chapter, we set the general context of this thesis. Our aim is to provide a comprehensive basis for the subsequent chapters. An overview of formal methods is presented in the first section. This is followed by a detailed account of Event-B [11, 14] and its toolset, Rodin [12]. The main concepts of Event-B are described with a particular emphasis on proof obligations. The Rodin platform is introduced in order to provide the practical setting of the contributions of this thesis. Furthermore, the sequent calculus used in Event-B reasoning is outlined together with the important notion of well-definedness. Next, we introduce three widely used formalisms in Isabelle/HOL [94, 90], PVS [91] and VDM [70, 69]. We conclude this chapter by conducting a brief comparative study between Event-B and the aforementioned formal methodologies.

2.1 Formal Methods

Mathematical techniques have a long important presence in all mature engineering disciplines. However, they have not been used as heavily in computer engineering [29, 110]. In fact, the debate about their use and relevance is an interesting one that has attracted considerable attention and is still doing so [74]. In [74], three schools of thought on this debate are singled out:

- One school of thought claims that formal techniques provide remedial and complete solutions to problems associated with system development.
- Another school of thought claims that formal methods have little use or benefit to the development process.
- A final school of thought considers formal methods to be *over-sold* and *under-used* according to [74].

We subscribe to the final school of thought without underplaying the importance of formal methods. It is vital to recognise that the complexity of computer systems is growing at a large rate, and as such there is an urgent need to have a systematic approach that can be employed to achieve adequate levels of dependability and trust in those systems.

Developing formal tools to reason about systems is indeed a challenging task. In [74], an interesting view on formal methods is presented. It lists the different components of which any formal method should consist:

- The *semantic model* is defined as the mathematical structure where terms, formulae and the rules used, are given a specific meaning. The semantic model should "reflect the underlying computational model of the intended application".
- The *specification language* is the notation with which systems and their behaviour are described. The specification language must "have a proper semantics within the semantic model".
- *Verification systems/refinement calculi* are the mathematically sound rules that allow the verification of system properties and the stepping between specifications and implementations.
- Supporting tools such as *proof assistants* and *syntax* and *type checkers* are important for the formalism to be of any practical use.

According to [74], a formal method should have clear development guidelines to facilitate its integration with development processes. The aim of this thesis is to enhance the existing Event-B verification system (by means of proof extensions) and specification language (by means of language extensions).

2.1.1 Challenges

Despite the availability of many formalisms and their supporting tools, there are many difficulties facing the integration of formal methods into the development process of computer systems. There are some real problems that stem from the very nature of formal methods and computer engineering. Some of these obstacles are outlined below and in [10]:

- Formal methods require computer engineers to think carefully about the system in question before proceeding to the coding stage. This is not helped by the fact that engineers "postpone any serious thinking" during the specification and design phases [14], and accommodate a rather long and resource-hungry test phase.

- It is quite difficult to change current practices with respect to the development process. Within the industry, managers are reluctant to change the traditional way of approaching projects unless a clear value will be gained.
- Modelling is not a simple activity as it is often accompanied by reasoning [10]. Clear distinction between modelling and programming should be attained as the initial model of a program specifies the properties against which the final program will be evaluated.
- One of the main objectives of modelling is the ability to reason formally. Software engineers are not accustomed to this practice.
- Finally, one of the main obstacles is the lack of appealing tool support to make modelling and reasoning a seamless addition to the development process. This is undoubtedly one of the main selling points of Event-B and Rodin [31, 12].

2.1.2 Classification

Despite the difficulties and misconceptions that surround formal methods, important efforts were spent designing and implementing formal systems and tools to benefit from the rigour that mathematics offer. In brief, these formalisms can be organised into five categories [74]:

1. *Model-based* approach: a system is modelled using discrete mathematical structures to describe its properties. Operations describe the transitions between different states. This approach does not explicitly represent concurrency. Non-functional requirements (e.g., temporal requirements) can, in some cases, be expressed. Notable examples of this approach include Z [109], the B Method [8] and VDM [70, 69].
2. *Logic-based* approach: logics are used to describe system properties including probabilistic and temporal behaviour. The axiomatic system of the used logic can then be employed to validate system properties. In some cases, the logic can be extended with concrete programming constructs to provide an implementation-oriented language. Notable examples of this approach include Modal Logic [56] and Temporal Logic [51].
3. *Algebraic* approach: In this approach, an explicit definition of operations is given by axiomatically linking the behaviour of different operations without defining states. Algebraic formalisms, similarly to model-based formalisms, do not provide an explicit representation of concurrency. A notable example of algebraic formalisms is OBJ [54].

4. *Process Algebra* approach: CSP [63] and CCS [86] are notable examples. The π-calculus [87] is a formal approach to model mobility within concurrent systems. Concurrent processes are formally represented, and system behaviours are described as "constraints on all allowable observable communication between processes" [87].

5. *Net-based* approaches: graphical notations with formal semantics are used to describe systems. Petri Nets [97] are a notable example.

Summary. In this section, we briefly discussed formal methods. We presented a number of challenges facing the adoption of formal methods in the industry. We concluded this section by outlining the different categories in which formal methodologies can be classified. The aim of this discussion was to provide a general context for the Event-B formalism and its toolset, Rodin.

2.2 Event-B

In this section, we give a brief account of Event-B. We start by describing what is meant by discrete systems which are the subject matter of Event-B modelling.

2.2.1 Discrete Systems Modelling

Complex systems are made of many inter-related components that interact with an external environment. Although these systems often exhibit continuous behaviours, they manifest discrete traits most of the time. This essentially means that they can be abstracted using a discrete transition model. There could be many of these transitions, but that does not change the very nature of such systems that are intrinsically discrete [14].

A discrete model consists of a *state* which can be represented as *variables*. The choice of variables will depend on the level of abstraction of the model with regard to the real system. Similarly to other applied sciences, there will be certain laws that should govern the state of the model including its type. Such laws are referred to as *invariants*.

A discrete model can be subject to a number of *transitions*, which we may refer to as *events*. Each of these events has a *guard* which is the condition under which the event is allowed to take place. Furthermore, each event has an *action* associated with it. The action describes the effect that the occurrence of the event has on the state of the model.

In the discrete modelling of complex systems, it is assumed that the execution of events takes no time [14]. When no event is allowed to occur (guards of all events are false), the execution of the model stops and is said to have *deadlocked* [14]. If many guards are

true, only one event is allowed to occur. The choice of the event to occur in the latter case is *non-deterministic*.

2.2.2 Event-B Modelling

Event-B is a formalism for discrete system modelling based on the B method [8]. Event-B modelling is carried out using first-order predicate logic with equality and set theory. The approach provides facilities to reason about models using proof obligations. These in turn implicitly represent the semantics of Event-B models [61]. In this subsection, we give a brief descriptive account of Event-B modelling. For a more detailed and formal description, see [11, 14].

An Event-B model consists of *contexts* and *machines*. Contexts represent the static aspects of the model whereas machines describe its dynamic aspects. Figure 2.1 summarises the anatomy of Event-B models.

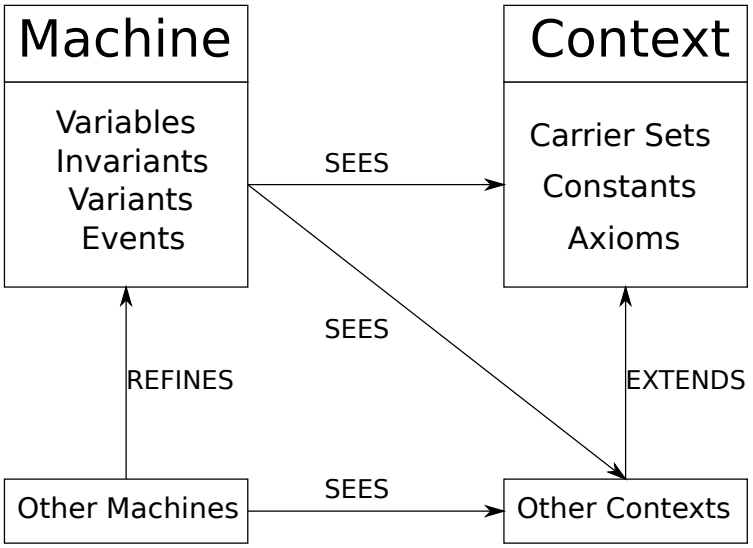


Figure 2.1: Anatomy of Event-B Models

2.2.2.1 Contexts

Contexts define static aspects of a model, and provide some of its axiomatic properties. They may contain *carrier sets*, *constants*, *axioms* and *theorems*. Carrier sets are assumed to be non-empty. Axioms are used to describe the properties of those sets and constants. Theorems are derived properties that should logically follow from the existing axioms. Proof obligations generated from contexts ensure that all axioms are *well-defined* and that all theorems are provable (i.e., logically follow from axioms) and well-defined. An axiom (or a theorem) is said to be well-defined if it does not contain ill-defined terms such as $x \div 0$. Finally, a context $C1$ can *extend* another context $C0$ (see Figure 2.1); this

CONTEXT DES_C
SETS
PERSON
CONSTANTS
age
minimumAge
AXIOMS
axm1 : $age \in PERSON \rightarrow \mathbb{N}$
axm2 : $minimumAge = 18$
END

Figure 2.2: Context **DES_C**

means that all carrier sets, constants, axioms and theorems defined in $C0$ are available to use in the axioms and theorems of $C1$. Figure 2.2 presents an Event-B context which defines a carrier set **PERSON** and two constants **age** and **minimumAge**. The following proof obligations are generated for contexts:

1. *Well-definedness of axiom* proof obligation to ensure that ill-defined terms are not present in axioms.
2. *Well-definedness of theorem* proof obligation to ensure that ill-defined terms are not present in theorems.
3. *Validity of theorem* to ensure that theorems are valid with respect to Event-B logic and any preceding axioms.

2.2.2.2 Machines

Machines provide the behavioural properties of Event-B models. They may contain *variables*, *invariants*, *theorems*, *variants* and *events*. Variables v define the state of a machine. Invariants $I(v)$ are constraints on variables v, and are similar to class invariants [84] in object-oriented languages. Class invariants are used to constrain objects of a particular class, and should not be violated by the execution of its methods. Similarly, machine invariants should not be violated by the execution of the events of the machine. Events describe possible state changes (i.e., *transitions*). Each event has a *guard* $G(t, v)$ and an *action* $S(t, v)$, where t are parameters of the event and v are the variables of the machine. The guard states the condition under which the event may occur. The action describes the effect of the occurrence of the event on the state of the machine. Contexts provide an independent placeholder for axiomatic properties, and machines can have access to these properties by means of a *sees* directive.

MACHINE DES_M
SEES DES_C
VARIABLES
`members`
`in`
INVARIANTS
`inv1` : $members \subseteq PERSONS$
`inv2` : $\forall m \cdot m \in members \Rightarrow age(m) \geq minimumAge$
`inv3` : $in \subseteq members$
EVENTS
Initialisation
begin
`act1` : $members := \varnothing$
`act2` : $in := \varnothing$
end
Event $addMembership \mathrel{\widehat{=}}$
any
m
where
`grd1` : $m \notin members$
`grd2` : $age(m) \geq minimumAge$
then
`act1` : $members := members \cup \{m\}$
end

Figure 2.3: Machine **DES_M**

An event `evt` can have one of the following three forms:

$$\texttt{evt} \mathrel{\widehat{=}} \textbf{begin } S(v) \textbf{ end} \tag{2.1}$$

$$\texttt{evt} \mathrel{\widehat{=}} \textbf{when } G(v) \textbf{ then } S(v) \textbf{ end} \tag{2.2}$$

$$\texttt{evt} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } G(t,v) \textbf{ then } S(t,v) \textbf{ end} . \tag{2.3}$$

where v are the variables of the machine, t are the parameters of the event,[1] $G(v)$ and $G(t,v)$ are the guards, and $S(v)$ and $S(t,v)$ are the actions.

Events of the form (2.1) do not have a guard, and as such can occur under all possible states of the system. A specialised event of the form (2.1) is used as an *initialisation* mechanism for state variables of the machine. Events of the form (2.2) have guards which restrict the state of the machine under which they can occur. In the final form (2.3), events have additional parameters, and their guards and actions are adjusted accordingly.

[1] Parameters of an event can be thought of as *local variables*.

The action of an event is defined in terms of several *generalised substitutions* (i.e., assignments) that can take one of the following three forms:[2]

$$x := E(t, v) \tag{2.4}$$

$$x :\in E(t, v) \tag{2.5}$$

$$x :| Q(t, v, x') \tag{2.6}$$

where $E(t, v)$ is an expression and $Q(t, v, x')$ is a predicate.[3]

Substitutions of the form (2.4) are *deterministic*. Substitutions of the other two forms are *nondeterministic*. Substitution (2.5) assigns x to an element of a set, whereas substitution (2.6) assigns x to a value satisfying the predicate $Q(t, v, x')$. Figure 2.3 presents an example Event-B machine.

The effect of each assignment can be described by means of a *before-after predicate* as follows:

$$BA(x := E(t, v)) \mathrel{\widehat{=}} x' = E(t, v) \tag{2.7}$$

$$BA(x :\in E(t, v)) \mathrel{\widehat{=}} x' \in E(t, v) \tag{2.8}$$

$$BA(x :| Q(t, v, x')) \mathrel{\widehat{=}} Q(t, v, x') \tag{2.9}$$

The before-after predicate ($BA(\cdot)$) describes the relationship between the state just before an assignment has occurred (represented by unprimed variable names x) and the state just after the assignment has occurred (represented by primed variable names x'). The assignment rule in Hoare logic [64] can be used to infer the weakest pre-condition in the case of an assignment, whereas the before-after predicate merely links the state of the machine before and after the execution of the event. Note that all assignments of an action occur simultaneously, therefore, a before-after predicate $A(t, v, x')$ for all assignments can be obtained by conjoining the before-after predicates of each individual assignment [60]. The machine variables y not appearing on the left hand side of an assignment remain unchanged. Finally, the before-after predicate of the action $S(t, v)$ can be written as follows:

$$BA(S(t, v)) \mathrel{\widehat{=}} A(t, v, x') \wedge y' = y \tag{2.10}$$

Following the same convention as in [60], we represent the before-after predicate of an action $S(t, v)$ by the predicate $\boldsymbol{S}(t, v, v')$.[4]

Proof obligations of machines are more involved than those of contexts, and serve to verify important properties. We use sequents to represent proof obligations for the

[2] If the event is parameterless, t can be removed from the left hand sides of the substitutions.

[3] Expressions and predicates are referred to as terms and formulae in some other literature.

[4] Note the bold faced $\boldsymbol{S}$ to differntiate the before-after predicate from the action $S(t, v)$.

remainder of this thesis. Sequents take the form $\boldsymbol{H} \vdash G$ where $\boldsymbol{H}$ is a set of hypotheses and G is the goal of the sequent.

Let an event `e` be defined according to (2.3), then the proof obligations are:

1. *Feasibility* proof obligation which ensures that the guard is the enabling condition of the event. Feasibility proof obligation is the following:

$$I(v), G(t, v) \ \vdash \ (\exists v' \cdot \boldsymbol{S}(t, v, v'))$$

2. *Invariant preservation* proof obligation which ensures that invariants hold whenever machine state changes. Invariant preservation proof obligation is the following:

$$I(v), G(t, v), \boldsymbol{S}(t, v, v') \ \vdash \ I(v')$$

2.2.2.3 Machine Refinement

Refining a machine makes the model more concrete. It is attained by refining both its state and events. The resulting machine has a state that is related to the state of the more abstract machine by a *gluing invariant*. The latter is expressed in terms of a predicate $J(v, w)$ linking the abstract state v and the refined state w. The refinement of events can take two shapes: refining existing events and introducing new ones.

Let **N** be a machine that refines another machine **M**, and let `aevt` and `cevt` be events in **M** and **N** respectively:

$$\texttt{aevt} \ \widehat{=} \ \textbf{any } t \textbf{ where } G(t, v) \textbf{ then } S(t, v) \textbf{ end} \tag{2.11}$$

$$\texttt{cevt} \ \widehat{=} \ \textbf{any } u \textbf{ where } H(u, w) \textbf{ then } T(u, w) \textbf{ end} \ . \tag{2.12}$$

Then, event `cevt` is said to refine event `aevt` if the following condition holds:

$$I(v), J(v, w), H(u, w), \boldsymbol{T}(u, w, w') \ \vdash \ \exists t. \, (G(t, v) \wedge \exists v'. \, (\boldsymbol{S}(t, v, v') \wedge J(v', w'))) \tag{2.13}$$

where $\boldsymbol{S}(u, w, w')$ and $\boldsymbol{T}(u, w, w')$ are the before-after-predicates associated with `aevt` and `cevt` respectively, $I(v)$ is the invariant of machine **M**, and $J(v, w)$ is the gluing invariant. In simple terms, a concrete eventb **cevt** is said to refine an abstract event **aevt** (1) when the guard of the former is stronger than the guard of the latter, and (2) when the gluing invariant is preserved by the conjoined action of both events [12].

Machine refinement can also introduce new events. Let `nevt` be a new event in machine **N**:

$$\texttt{nevt} \ \widehat{=} \ \textbf{any } u \textbf{ where } H(u, w) \textbf{ then } T(u, w) \textbf{ end} \tag{2.14}$$

then the following three additional conditions must hold to ensure that machine **N** is a valid refinement of machine **M**:

1. Event `nevt` must refine an implicit event in the abstraction **M** that does nothing (`skip`). This leads to the following proof obligation:

$$I(v), J(v,w), H(u,w), \boldsymbol{T}(u,w,w') \quad \vdash \quad J(v,w') \tag{2.15}$$

2. Event `nevt` must not *diverge* (run forever) since, otherwise, it would make previously enabled abstract events effectively disabled. Formally:

$$I(v), J(v,w), H(u,w), \boldsymbol{T}(u,w,w') \quad \vdash \quad V(w') < V(w) \tag{2.16}$$

   where $V(w)$ and $V(w')$ are expressions over the set of natural numbers[5]. V is called a *variant*, and its value is decreased by each new event. The variant is an expression that is supplied by the modeller as part of the machine (see Figure 2.1).

3. The concrete machine **N** must not *deadlock* before its abstraction **M** for, otherwise, **N** might not achieve what **M** required. Formally:

$$I(v), J(v,w), (G_1(v) \vee ... \vee G_n(v)) \quad \vdash \quad (H_1(w) \vee ... \vee H_m(w)) \tag{2.17}$$

   where $G_i(v)$ are the guards in the abstraction **M**, and $H_j(w)$ are the concrete guards.

2.2.3 Event-B Pragmatics

The Event-B modelling notation has been designed to be "simple and easily teachable" [60]. It is targeted at modelling complex systems, and as such tool support is a major aspect of its appeal. In what follows, we briefly outline some of the important choices made when designing the Event-B notation as discussed by Hallerstede in [60]:

1. *Modelling versus Programming*: Important choices regarding modelling and programming were made when conceiving the Event-B notation. Hallerstede claims modelling and programming are seen as activities of different nature with varying objectives. A program can be executed, whereas execution is not required for a model. As such, many traits of programming languages have been omitted in order to reduce the complexity of the notation and put more emphasis on reasoning. However, this may increase the efforts needed to specify certain aspects of systems including sequencing.

[5] Variants can be more elaborate, see [14].

- *Sequential Composition*: Sequential composition can complicate proof obligations and make them difficult to comprehend, and as such Event-B does not support them.
- *Conditional Statements*: These are not supported in Event-B. Conditional statements pose a significant challenge when proving refinement proof obligations, as it is not easy to work out which branches in the refinement correspond to which branches in the abstraction. Instead, Event-B adopts an approach whereby each branch corresponds to a separate event.

2. *Undefinedness*: Conditionally defined expressions are frequently used when developing models. This poses a major challenge when the underlying logic is the two-valued first order logic. To deal with this issue, Event-B considers the well-definedness of expressions at the level of type-checking. Type-checking works in two passes. The first pass checks whether expression are correctly typed regardless of whether they are defined. The second pass of the type-checker creates well-definedness proof obligations that must be discharged by proof [60]. For example, the expression $1 \div 0$ is correctly typed, but is not well-defined as it cannot be shown that $0 \neq 0$.

3. *Parameterisation*: Models can depend on many parameters, e.g., number of components in a structure. Event-B contexts are used to parameterise machines using carrier sets and constants. These can be instantiated, and if they satisfy the axioms of the context, the theorems derived from them can be readily used.

4. *Openness*: The Event-B modelling notation is not finalised, and is expected to evolve according to the different needs and application domains. The formalism is open to extensions and changes. Hallerstede emphasises, however, that care should be taken to avoid complicating the existing theory, and concepts should be interpreted in a simple and unambiguous way [60].

Summary. In this section, we presented a brief account of Event-B. We started by providing an overview of discrete systems modelling. Next, contexts and machines were discussed as well as their proof obligations. The important concept of machine refinement is presented. We concluded this section by presenting an overview of the different choices made when designing the Event-B modelling notation as discussed by Hallerstede in [60].

2.3 The Rodin Platform

The Rodin platform [12, 31] is an *integrated modelling environment* for Event-B. It provides facilities and tools to develop and reason about models in a *reactive* manner inspired by modern integrated development environments (IDEs) such as Eclipse [49]. When developing Java programs using Eclipse, the user is not required to initiate the

compilation process. Rather, the IDE reacts to changes in code in a seamless manner which provides an effective feedback to the developer. Analogously, in Rodin, while developing a model of a complex system, static checking, proof obligations generation and management are carried out seamlessly to provide immediate feedback to the modeller. The combination of static checking and proof obligation generation in Rodin can be thought of as an extended static checker [44] for Event-B. More precisely, the Rodin platform provides the capabilities to:

- develop models in Event-B by specifying contexts and machines,
- analyse models by means of static checking which includes syntax and type checking,
- semantically analyse models by means of proof obligations generated as appropriate,
- carry out mathematical proof in order to verify model consistency.

In order to strike a good balance between usability and effectiveness, Rodin is designed to satisfy the following requirements [31]:

- "Design-Time Feedback": the tool responds quickly to changes and provides feedback that can be easily related to models;
- "Distinct Proof Obligation Generation and Verification phases": the tool decouples modelling and proving while maintaining the link between the two activities (i.e., traceability) in case automatic proofs fail.

2.3.1 Architecture

Figure 2.4 shows a high-level view of the internal architecture of Rodin. The tool can be divided into four distinct components which are described below:

1. *The Rodin Core*: contains the *Rodin repository* and the *Rodin builder*. The repository manages the persistence between data elements (Java objects, e.g., proof obligations) and their storage in XML files (e.g,. proof obligation files). The builder (analogous to the Java builder in the Eclipse Java IDE) schedules jobs depending on changes to files in the repository.
2. *The Event-B Library Packages*: the syntax of the Event-B mathematical language is specified by an attributed grammar implemented in the *abstract syntax tree* (AST) module. The *sequent prover* (SEQP) module provides the necessary infrastructure to carry out proofs.

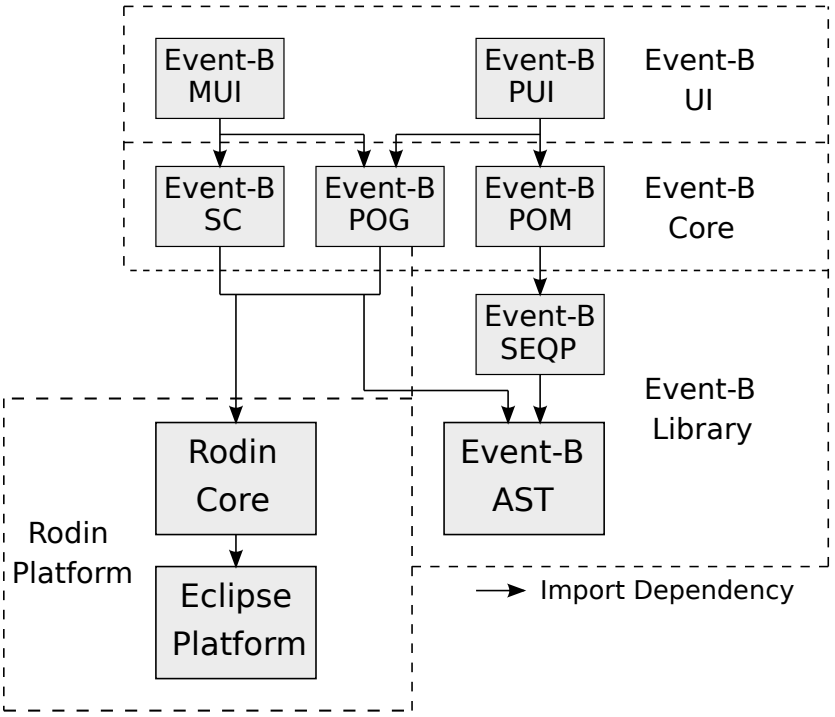


Figure 2.4: Rodin Tool Architecture

3. *The Event-B Core*: contains the *static checker* (SC), the *proof obligation generator* (POG) and the *proof obligation manager* (POM). The static checker analyses contexts and machines in terms of syntax as well as typing. The proof obligation generator generates proof obligations from statically checked elements of the model including axioms, theorems, invariants and events. Finally, the proof obligation manager keeps track of proof obligations and their proofs.

4. *The Event-B User Interface*: contains the graphical interactivity model for Event-B. It provides two distinct perspectives: the *modelling user interface* (MUI) and the *proving user interface* (PUI).

Figure 2.5 describes the tool-chain for developing Event-B models using the Rodin platform.

2.3.2 The Rodin Tooling Philosophy

Modelling is a complex activity, and is a hugely important step in developing complex and reliable systems. Reasoning can significantly improve understanding of a particular model. An effective tool support should provide a practical setting for creating models and reasoning about them. It also should make the transition required between the modelling and reasoning activities as seamless as possible.

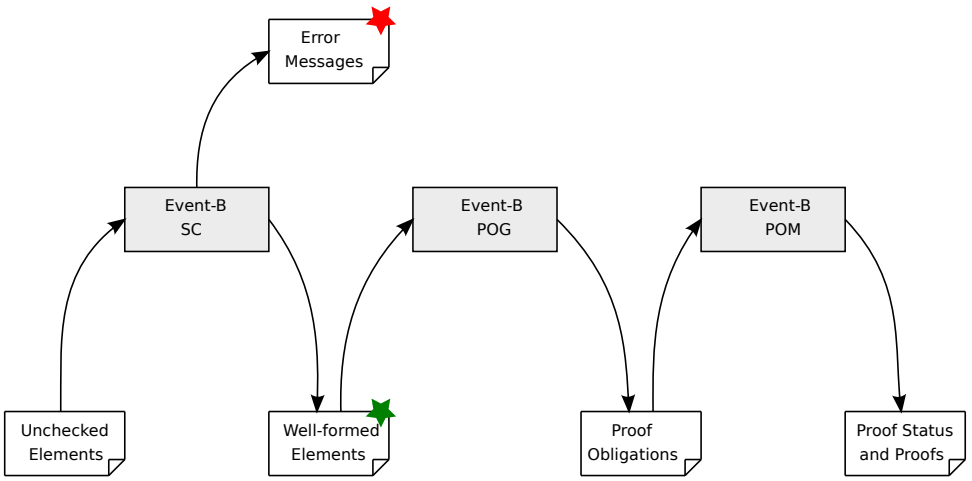


Figure 2.5: Rodin Tool-Chain for Event-B

Hallerstede [60] provides an overview of the different choices and decisions (some of which are summarised in §2.2.3) made when conceiving the notation and the modelling environment for Event-B. Moreover, the Event-B toolset (Figure 2.6) aims to satisfy the following two requirements [12]:

1. "Design-Time Feedback": The tool is very responsive and immediately provides feedback related to the model. A further requirement is that the feedback should easily relate to the model in question.

2. "Distinct Proof Obligation Generation and Verification Phase": This is important as it allows the user to distinguish between the modelling and proving activities. This is particularly important when proofs fail, as it allows the origin of the proof obligation to be traced more easily.

2.3.2.1 Editors

The Rodin platform provides editors for contexts and machines. The editors are designed to mirror the structure of their respective files. Since context and machine files have an XML structure, their respective editors have a tree look, and are form-based[6]. There is a text-based editor for Rodin called Camille [4]. However, this editor suffers from several bugs that hinder its usability.

2.3.2.2 Tooling

Tooling refers to the collection of tools that run on Rodin files. Figure 2.5 describes the three tools available in the Rodin repertoire. The Rodin tool chain refers to the different stages of tooling:

[6]This particular design decision was taken to account for possible extensions to the Rodin database.

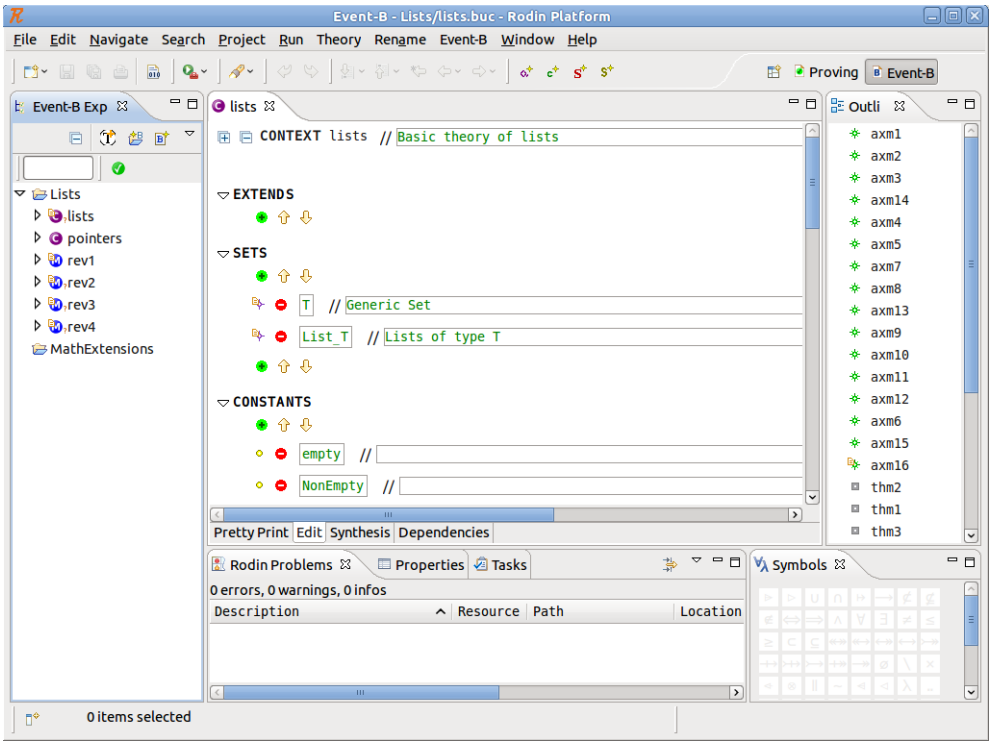


Figure 2.6: The Rodin Tool

1. **Static Checking**. Event-B components (i.e., contexts and machines) are statically checked for syntax and typing errors. Each Rodin file has two versions: 1) an unchecked version, and 2) a statically checked version. The unchecked file is the version that can be edited by the user. Unchecked machines and contexts have file extensions '.bum' (i.e., B unchecked machine) and '.buc' (i.e., B unchecked context) respectively. Checked machines and contexts have file extensions 'bcm' (i.e., B checked machine) and 'bcc' (i.e., B checked context) respectively. The purpose of the static checker is to create the static checked files ('.bcm' and '.bcc') from their unchecked counterparts ('.bum' and '.buc'), and in the process eliminating any ill-formed elements. The static checker goes through all the sub-elements of the unchecked file, and generates their statically checked counterparts if all the required conditions are met by each element. The statically checked files are, then, the subject of subsequent tooling.

2. **Proof Obligation Generation**. This refers to the generation of proof obligations from the well-formed elements of contexts and machines. Obligation generation runs on statically checked contexts and machines. The proof obligations generated in Rodin are presented in §2.2.2, and are more elaborately justified in [60].

3. **Proof Management**. This refers to the management of the relationship between proof obligations and their proofs. A proof obligation can be: 1) pending, 2) discharged, or 3) reviewed. A proof obligation is reviewed if it has been inspected

by the user and is ear-marked to be discharged later. The state of a proof obligation is determined by the state of its proof (i.e., complete or incomplete). The Rodin prover alters the state of a proof by 1) applying proof rules, or 2) invoking external provers (ML, PP [6] and more recently Event-B Isabelle prover [101]).

2.3.2.3 Reactive Development

The Rodin platform proposes a reactive modelling environment [80, 82] similar to modern integrated development environments (IDE's), hence the decision to implement Rodin on top of the Eclipse IDE. The user working on a model is constantly updated on the status of her/his proofs. To achieve this, the tools in Rodin repertoire run in a reactive manner by:

1. checking models for syntax and type errors,
2. generating proof obligation where appropriate, and
3. updating the status of its proofs by calling automated provers, or reusing old proof attempts [82].

The reactive nature of Rodin poses many challenges with respect to proofs. Mehta [82] outlines the different issues and his approach to dealing with them (proof reuse and re-engineering.

2.3.2.4 Proof Obligations

Proof obligations are central to Event-B modelling. The naming of proof obligations and their structure is crucial to facilitating the modelling activity [60]. Proof obligations are easily traceable to their corresponding element in contexts and machines, making the transition between modelling and proof easier.

2.3.3 Event-B Mathematical Language

Figure 2.7 shows an example of a simple context. Context C0 defines a constant *minimum*. The first axiom asserts that constant *minimum* is a partial function from the set of sets of naturals to the set of naturals. The second axiom ensures that *minimum* associates non-empty sets of natural numbers with their least element using the usual ordering $\leq$ on natural numbers. The syntax used to write Event-B models can be decomposed into two levels:

CONTEXT C0
CONSTANTS
minimum
AXIOMS
axm1 : $minimum \in \mathbb{P}(\mathbb{N}) \pfun \mathbb{N}$
axm2 : $\forall s \cdot (s \in \mathbb{P}(\mathbb{N}) \land s \neq \varnothing) \Rightarrow (\forall n \cdot n \in s \Rightarrow minimum(s) \leq n)$
END

Figure 2.7: Event-B Outer and Inner Syntax

1. *Outer Syntax*: this level of syntax corresponds to the unboxed parts of the context definition in Figure 2.7. This syntax is used to specify the components of individual contexts and machines.

2. *Inner Syntax*: this level of syntax corresponds to the boxed parts in Figure 2.7. This syntax is used to specify the mathematical formulae corresponding to axioms, invariants, guards and actions.

The inner syntax of Event-B is specified by means of an attributed grammar, and is defined in the (AST) sub-module of Rodin, see §2.3.1. The outer syntax, on the other hand, is specified by a database of elements whose relationships are specified by a graph. Thanks to the Rodin database [12, 60], the outer syntax is easily extensible. This facilitated the development of several useful plug-ins, e.g., the Modularisation plug-in [66] and the Records plug-in [104].

The inner syntax, prior to Rodin version 2.0, was wired in the (AST) sub-module, and could not be extended as easily as the outer syntax. However, Rodin 2.0 provided a dynamic parser for the inner syntax which can be easily augmented with new syntax [3]. From hereon, we shall refer to the inner syntax as the *mathematical language* of Event-B [83]. The mathematical language is the level of syntax whose extensibility is addressed by the contributions of the thesis. In particular, two important aspects of extensions are considered: practicality of use and soundness.

2.3.4 Proof Infrastructure

The proof obligation manager (POM), described in §2.3.1, manages the relationship between proof obligations and their proofs. The *proof manager* (PM) is in charge of handling and maintaining proofs, and provides important services to POM. For each proof obligation, it constructs a proof tree whose root is the sequent of the obligation itself. The proof manager works both automatically (without user intervention) and interactively (with user intervention and possibly with input).

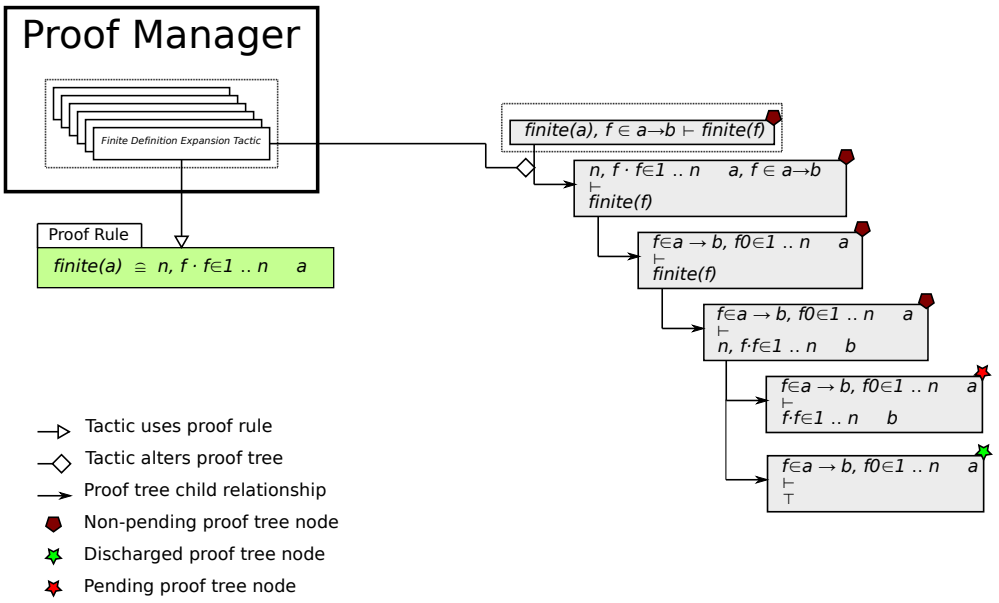


Figure 2.8: The Proof Manager (PM)

A detailed description of Rodin's prover architecture is described by Mehta in his thesis [82]. We summarise the key elements of the architecture:

- *Proof Trees* are recursive structures based on proof tree nodes. A proof tree node represents a single node as well as the proof tree (or sub-tree) rooted at that node, see Figure 2.8. Each proof tree node has a sequent. It may also have *a justifying proof rule* and a list of child nodes. A proof tree node can be either:

  1. *pending*, if its proof rule is `null`, consequently, the list of child nodes is `null`, or,
  2. *non-pending*, if it has a `non-null` proof rule, and the child nodes correspond to the result of applying the proof rule to its sequent.

- *Tactics* were introduced by Robin Milner in the early 1970 for the LCF theorem prover [85]. They provide a uniform mechanism to manipulate proof trees. A tactic could be a wrapper around a proof rule in which case it is called a basic tactic. Tactical tactics, on the other hand, are more structured and can be used to specify a proof strategy [82]. An example is a tactic that repeats another tactic until it fails.

- *Reasoners* are concrete proof rule generators. An example proof rule is the following well-documented conjunction-introduction rule:

$$\frac{\mathbf{H} \vdash P \quad \mathbf{H} \vdash Q}{\mathbf{H} \vdash P \wedge Q} \wedge intro$$

Concrete proof rules can be generated by appropriately instantiating the meta-variables $\mathbf{H}$,[7] P and Q. Using a simple Java-like language, Listing 2.1 describes the general interface reasoners obey [82]:

The `apply` method checks whether the proof rule is applicable to the given sequent with the supplied input [8], and if so generates a concrete proof rule which will be the justification for the proof step. If the rule is not applicable, no change occurs in the proof tree.

```
interface Reasoner{
        Rule apply(Sequent sequent, ReasonerInput input);
}

interface ReasonerInput{}
```

Listing 2.1: Reasoner Protocol

The proof manager can be extended with new reasoners and tactics. There is a well-defined protocol for both extensions. Reasoners are also used to integrate external provers. The idea is to encapsulate a call to the external prover as a reasoner application. The call is successful if the external prover discharges the sequent, i.e., if it finds a complete proof for the sequent. One limitation is that information about how the external prover went about the proof (e.g., used hypotheses) is not always available to the proof manager.

Two external provers that have been successfully integrated are:

1. *The Predicate Prover (PP)*: this prover is built around a hierarchy of provers. It contains a decision procedure for propositional logic and a semi-decision procedure for first order logic. Another major component is the translator from set theory to first order logic. It is built in accordance with the set-theoretic construction outlined in the B Book [8].
2. *The ML Prover (ML)*: is a rule-based prover used in the Logic Solver which is the compiler-interpreter used for B. PP was originally developed to validate the many proof rules of ML. ML and PP are part of Atelier-B [6] which provides the proving infrastructure for B.

Despite being optimised for proof reuse [82], the current architecture has the following limitations:

- in order to add a new proof rule, it was required to implement a reasoner and a wrapper tactic. Therefore, a certain level of competence with the Java programming language as well as knowledge of Rodin architecture were necessary;

[7] Note that $\mathbf{H}$ stands for a set of formulae (the set of hypotheses)
[8] Input could, for example, be a term to instantiate a universally quantified formula.

- after a new rule is added, soundness of the prover augmented with the new rule has to be established. It is not clear how this can be achieved at the level of Java code. The use of Java verification tools, e.g., JML [30] has not been adopted by Rodin as of the time of writing this thesis.

Summary. The aim of this section was to provide the practical setting of the contributions of this thesis. We presented an overview of the Rodin platform. The general architecture of the toolset is discussed. A particular focus is placed on the tooling aspects of Rodin including static checking, proof obligation generation and proof management. We also provided a brief description of the Event-B mathematical language and proof infrastructure.

2.4 Reasoning in Event-B

In this section, we define the mathematical logic that will be used in the proof system of Event-B. We also discuss in detail the proof calculus employed in Event-B reasoning. The important notion of well-definedness is thoroughly studied, and its link to partial functions is presented. The mathematical logic defined herein will also be used in Chapter 3.

2.4.1 First-order Predicate Calculus with Equality

In the next two definitions, we introduce the syntax of the first-order predicate calculus with equality. We use the language signature Σ defined by a set V of variable symbols, a set F of function symbols and a set P of predicate symbols. In line with [8, 11, 25, 81, 82], we distinguish between terms and formulae.

Definition 2.1 (Term)**.** T_Σ, the set of Σ-terms, is inductively defined as follows:
• each variable of V is a term;
• if $f \in F$, $arity(f) = n$ and each of $e_1, ..., e_n$ is a term, then $f(e_1, ..., e_n)$ is a term.

Definition 2.2 (Formula)**.** F_Σ, the set of Σ-formulae is inductively defined as follows:
• $p(t_1, ..., t_n)$ is a formula provided $p \in P$, $arity(p) = n$ and each of $t_1, ..., t_n$ is a term;
• $t_1 = t_2$ is a formula provided t_1 and t_2 are terms;
• $\bot$ is a formula;
• $\varphi \wedge \psi$ is a formula if φ and ψ are formulae;
• $\neg\varphi$ is a formula if φ is a formula;
• $\forall x.\varphi$ is a formula if $x \in V$ and φ is a formula.

We also use other (standard) logical operators defined using the following syntactic definitions:

$$\begin{array}{rcl} \top & \widehat{=} & \neg\bot \\ \varphi \vee \psi & \widehat{=} & \neg(\neg\varphi \wedge \neg\psi) \\ \varphi \Rightarrow \psi & \widehat{=} & \neg\varphi \vee \psi \\ \varphi \Leftrightarrow \psi & \widehat{=} & (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi) \end{array}$$

In the absence of potentially ill-defined terms, the previous first-order language can be assigned a two-valued semantics. In this case, the sequent calculus **LK** [50] can be used for conducting proofs.

2.4.2 Defining Partial Functions

In this section, we show how a partial function can be added as a definitional extension by means of a *conditional definition* [15, 81]. A partial function symbol f is introduced using the following conditional definition:

$$\frac{}{C^f_{\vec{x}} \;\vdash\; y = f(\vec{x}) \;\Leftrightarrow\; D^f_{\vec{x},y}}\; f_{def}$$

which can be added as an axiom to the proof theory of the previous first-order language, provided [81]:

1. Variable y is not free in $C^f_{\vec{x}}$,
2. Formula $D^f_{\vec{x},y}$ only contains the free variables from $\vec{x}$ and y,
3. Formulae $C^f_{\vec{x}}$ and $D^f_{\vec{x},y}$ only contain previously defined symbols,
4. The following theorems:
   - **Uniqueness:** $C^f_{\vec{x}} \;\vdash\; \forall y, z \cdot (D^f_{\vec{x},y} \wedge D^f_{\vec{x},z}) \;\Rightarrow\; y = z$
   - **Existence:** $C^f_{\vec{x}} \;\vdash\; \exists y \cdot D^f_{\vec{x},y}$

   must be provable from the existing theory and any previously introduced definitions.

The above definition meets the two criteria of a definitional extension: *Criterion of Eliminability* and *Criterion of Non-creativity* [15, 105]. The formula $C^f_{\vec{x}}$ is the *well-definedness condition* of f which effectively defines its domain. For a total function symbol, the well-definedness condition is $\top$. In the case where $C^f_{\vec{x}}$ holds, the conditional definition f_{def} can be used to eliminate all occurrences of f in a term or formula by its

definition $D^f_{\vec{x},y}$. As an example, we consider (conditionally) defining the infix division function in a theory of real numbers:

$$\frac{}{y \neq 0 \ \vdash \ z = x \div y \Leftrightarrow x = z \times y} \ \div_{def}$$

This definitions allows '$\div$' to be unfolded when its second argument is not equal to 0. The term $1 \div 0$ is syntactically acceptable, but is said to be *ill-defined*. In the classical sense, the formula $1 \div 0 = 1 \div 0$ can be shown to be valid on the basis of their logical structure [81]. As such, the classical first-order sequent calculus (such as **LK**) is not a suitable proof calculus as it does not account for ill-defined terms.

2.4.3 The Well-Definedness Operator

The well-definedness operator '$\mathcal{D}$' formally encodes what is meant by well-definedness. $\mathcal{D} : (F_\Sigma \cup T_\Sigma) \rightarrow F_\Sigma$ is a syntactic operator that maps terms and formulae to their well-definedness conditions (which are themselves formulae). We interpret the formula $\mathcal{D}(F)$ as being valid if and only if F is well-defined. For a detailed treatment of the $\mathcal{D}$ operator, we refer to [15].

The well-definedness (WD) of terms is defined recursively as follows:

$$\mathcal{D}(x) \ \widehat{=} \ \top \quad if \ x \in V \tag{2.18}$$

$$\mathcal{D}(f(t_1, ..., t_n)) \ \widehat{=} \ \bigwedge_{i=1}^{n} \mathcal{D}(t_i) \ \wedge \ C^f_{t_1,...,t_n} \tag{2.19}$$

where $C^f_{t_1,...,t_n}$ effectively defines the domain of the function f. For this study, we assume that predicate symbols are total. As a result, ill-definedness can only be introduced by terms. Therefore, we have the following:

$$\mathcal{D}(p(t_1, ..., t_n)) \ \widehat{=} \ \bigwedge_{i=1}^{n} \mathcal{D}(t_i) \quad if \ p \in P \tag{2.20}$$

$$\mathcal{D}(t_1 = t_2) \ \widehat{=} \ \mathcal{D}(t_1) \wedge \mathcal{D}(t_2) \tag{2.21}$$

For the well-definedness of other formulae, we use the following expansions from [15]:

$$\begin{aligned}
\mathcal{D}(\bot) &\ \widehat{=} \ \top \\
\mathcal{D}(\neg\varphi) &\ \widehat{=} \ \mathcal{D}(\varphi) \\
\mathcal{D}(\varphi \wedge \psi) &\ \widehat{=} \ (\mathcal{D}(\varphi) \wedge \mathcal{D}(\psi)) \vee (\mathcal{D}(\varphi) \wedge \neg\varphi) \vee (\mathcal{D}(\psi) \wedge \neg\psi) \\
\mathcal{D}(\forall x \cdot \varphi) &\ \widehat{=} \ (\forall x \cdot \mathcal{D}(\varphi)) \vee (\exists x \cdot \mathcal{D}(\varphi) \wedge \neg\varphi)
\end{aligned}$$

Well-definedness conditions related to the derived logical operators can be easily obtained, e.g.,

$$\mathcal{D}(\varphi \lor \psi) \quad \Leftrightarrow \quad (\mathcal{D}(\varphi) \land \mathcal{D}(\psi)) \lor (\mathcal{D}(\varphi) \land \varphi) \lor (\mathcal{D}(\psi) \land \psi)$$

Intuitively, the above definitions enumerate all the possible conditions under which a formula can be *evaluated*. In the case of disjunction, the formulae can be evaluated if:

1. both disjuncts are well-defined; or
2. either one of the disjuncts is well-defined and is evaluated to **true**.

Semantic treatment of the $\mathcal{D}$ operator can be found in [25, 16].

An important property of well-definedness conditions is that they are themselves well-defined as shown in [81]; i.e.,

$$\mathcal{D}(\mathcal{D}(P)) \;\Leftrightarrow\; \top \tag{2.22}$$

For the remainder of this thesis, we mainly use the $\mathcal{D}$ operator. We may also refer to another well-definedness operator $\mathcal{L}$ [15]. Well-definedness conditions generated by means of operator $\mathcal{L}$ are smaller in size compared to their $\mathcal{D}$ counterparts. For this particular reason, the Rodin platform employs $\mathcal{L}$ as it makes proofs less tedious to perform. For the different logical operators, we have the following $\mathcal{L}$-generated well-definedness conditions:

$$\mathcal{L}(\neg\varphi) \quad \widehat{=} \quad \mathcal{L}(\varphi) \tag{2.23}$$

$$\mathcal{L}(\varphi \land \psi) \quad \widehat{=} \quad \mathcal{L}(\varphi) \land (\varphi \Rightarrow \mathcal{L}(\psi)) \tag{2.24}$$

$$\mathcal{L}(\forall x \cdot \varphi) \quad \widehat{=} \quad \forall x \cdot \mathcal{L}(\varphi) \tag{2.25}$$

The following property asserts that $\mathcal{L}$ is stronger than $\mathcal{D}$:

$$\mathcal{L}(\varphi) \;\Rightarrow\; \mathcal{D}(\varphi) \tag{2.26}$$

Property 2.26 can be shown by structural induction on φ. It merely states that if a formula is shown to be well-defined with respect to $\mathcal{L}$, it will also be well-defined with respect to $\mathcal{D}$. This is particularly useful in Rodin as the use of $\mathcal{L}$ greatly simplifies proofs of well-definedness. There is, however, a compromise on completeness due to $\mathcal{L}$ being sensetive to the order of formulae.

2.4.4 Well-Definedness and Proof

In this section, we explain the approach taken in reasoning with Event-B when dealing with ill-defined terms. The notion of well-definedness can be integrated into a classical

$$\frac{}{H, P \vdash P}\ hyp \qquad \frac{H \vdash Q}{H, P \vdash Q}\ mon \qquad \frac{H, \neg Q \vdash \bot}{H \vdash Q}\ contr$$

$$\frac{}{H, \bot \vdash P}\ \bot hyp \qquad \frac{H \vdash P}{H, \neg P \vdash Q}\ \neg hyp \qquad \frac{H \vdash P \quad H \vdash Q}{H \vdash P \land Q}\ \land goal$$

$$\frac{H, P, Q \vdash R}{H, P \land Q \vdash R}\ \land hyp \qquad \frac{H \vdash P}{H \vdash \forall x \cdot P}\ \forall goal\ (x\ \underline{\text{nfin}}\ H) \qquad \frac{}{H \vdash E = E}\ = goal$$

$$\frac{H \vdash [x := E]P}{H, E = F \vdash [x := F]P}\ = hyp \qquad \frac{H \vdash P \quad H, P \vdash Q}{H \vdash Q}\ cut$$

$$\frac{H, [x := E]P \vdash Q}{H, \forall x \cdot P \vdash Q}\ \forall hyp$$

Figure 2.9: Inference Rules of ***FoPCe*** [81]

first-order sequent calculus to obtain a proof calculus that is suitable for handling partial functions.

The aim of Mehta's work is to use the classical sequent calculus (see Figure 2.9) in Event-B proofs, and as such, the notion of validity cannot be changed. Instead, a pragmatic approach, in which validity and well-definedness are separated, is taken. To avoid ill-defined proof obligations being discharged, both validity and well-definedness are required to hold [81]. For example, the sequent $\vdash 1 \div 0 = 1 \div 0$ is allowed to be proven to be valid. However, it cannot be proved to be well-defined. When proving a proof obligations $H \vdash G$, we are obliged to prove two proof obligations:

$$\boxed{\text{WD}: \ \vdash \mathcal{D}(\mathbf{H} \vdash G)} \qquad \boxed{\text{Validity}: \ \mathbf{H} \vdash G}$$

The first proof obligation, WD, is the well-definedness proof obligation, and is expressed using the well-definedness operator $\mathcal{D}$ that was introduced in §2.4.3, and is defined for sequents in §2.4.4.1. The second proof obligation, Validity, is the validity proof obligation. Note that both proof obligations, WD and Validity, can be proved using ***FoPCe*** [81].

Proving well-definedness can be seen as filtering out formulae that contain ill-defined terms. In the case of $\vdash 1 \div 0 = 1 \div 0$, we are also required to prove $\vdash 0 \neq 0 \land 0 \neq 0$ as its WD (this proof obligation is obtained using the definition 2.21). Since this is not provable, we have filtered out the sequent $\vdash 1 \div 0 = 1 \div 0$ as not being well-defined in the same way we would have filtered out $\vdash 1 = \{1\}$ as not being well-typed [81]. Unlike type-checking, well-definedness is undecidable, and requires mathematical proof.

When proving the validity of a sequent, it can be assumed to be well-defined (as there is a separate proof obligation to ensure well-definendness). However, only the initial sequent of Validity can be assumed to be well-defined. In order to take advantage of

the property of well-defined sequents across proofs, we can only use proof rules that preserve well-definedness [81]. In §2.4.4.2, we present a proof calculus that preserves well-definedness across proofs.

2.4.4.1 Well-Defined Sequents

The $\mathcal{D}$ operator can be extended for sequents as follows:

$$\mathcal{D}(\mathbf{H} \vdash G) \;\widehat{=}\; \mathcal{D}(\forall \vec{x} \cdot \bigwedge \mathbf{H} \Rightarrow G) \tag{2.27}$$

where the following conventions are used:

- $\mathbf{H}$ is a finite sequence of formulae,
- $\bigwedge \mathbf{H}$ denotes the conjunction of all formulae present in $\mathbf{H}$,
- $\forall \vec{x}$ denotes the universal quantification of all free variables occurring in $\mathbf{H}$ and G.

A sequent $\mathbf{H} \vdash G$ is said to be well-defined if we can additionally assume that $\mathcal{D}(\mathbf{H} \vdash G)$ is present in its hypotheses [81]. The syntactic sugar $\vdash_{\mathcal{D}}$ is used to denote well-defined sequents:

$$\mathbf{H} \vdash_{\mathcal{D}} G \quad \widehat{=} \quad \mathcal{D}(\mathbf{H} \vdash G), \mathbf{H} \vdash G$$

Examples. Consider the following two sequents:

$$\begin{aligned} x = 1 \quad &\vdash_{\mathcal{D}} \quad x = 1 \\ &\vdash_{\mathcal{D}} \quad 1 \div 0 = 1 \end{aligned}$$

The previous two sequents are equivalent to:

$$\begin{aligned} \mathcal{D}(x = 1),\ x = 1 \quad &\vdash \quad x = 1 \\ \mathcal{D}(1 \div 0) \quad &\vdash \quad 1 \div 0 = 1 \end{aligned}$$

Furthermore, the previous sequents can be simplified further to:

$$\begin{aligned} x = 1 \quad &\vdash \quad x = 1 \\ 0 \neq 0 \quad &\vdash \quad 1 \div 0 = 1 \end{aligned}$$

Note that the sequent '$x = 1 \vdash x = 1$' is well-defined, since the well-definedness of both the goal and the hypothesis evaluate to $\top$, and hence implicitly present in the hypotheses. Therefore, the two sequents '$x = 1 \vdash x = 1$' and '$x = 1 \vdash_{\mathcal{D}} x = 1$' are equivalent. However, the two sequents '$\vdash_{\mathcal{D}} 1 \div 0 = 1$' and '$\vdash 1 \div 0 = 1$' are not equivalent.

$$\frac{}{H, P \vdash_{\mathcal{D}} P}\ hyp_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} Q}{H, P \vdash_{\mathcal{D}} Q}\ mon_{\mathcal{D}} \qquad \frac{H, \neg Q \vdash_{\mathcal{D}} \bot}{H \vdash_{\mathcal{D}} Q}\ contr_{\mathcal{D}}$$

$$\frac{}{H, \bot \vdash_{\mathcal{D}} P}\ \bot hyp_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} P}{H, \neg P \vdash_{\mathcal{D}} Q}\ \neg hyp_{\mathcal{D}} \qquad \frac{H, P \vdash_{\mathcal{D}} \bot}{H \vdash_{\mathcal{D}} \neg P}\ \neg goal_{\mathcal{D}}$$

$$\frac{H \vdash_{\mathcal{D}} P \quad H \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} P \wedge Q}\ \wedge goal_{\mathcal{D}} \qquad \frac{H, P, Q \vdash_{\mathcal{D}} R}{H, P \wedge Q \vdash_{\mathcal{D}} R}\ \wedge hyp_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} P}{H \vdash_{\mathcal{D}} \forall x \cdot P}\ \forall goal_{\mathcal{D}}\ (x\ \underline{\text{nfin}}\ H)$$

$$\frac{H \vdash_{\mathcal{D}} [x := E]P}{H, E = F \vdash_{\mathcal{D}} [x := F]P} = hyp_{\mathcal{D}} \qquad \frac{\boxed{H \vdash_{\mathcal{D}} \mathcal{D}(P)} \quad H \vdash_{\mathcal{D}} P \quad H, P \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} Q}\ cut_{\mathcal{D}}$$

$$\frac{}{H \vdash_{\mathcal{D}} E = E} = goal_{\mathcal{D}} \qquad \frac{\boxed{H \vdash_{\mathcal{D}} \mathcal{D}(E)} \quad H, [x := E]P \vdash_{\mathcal{D}} Q}{H, \forall x \cdot P \vdash_{\mathcal{D}} Q}\ \forall hyp_{\mathcal{D}}$$

Figure 2.10: Inference Rules of $\boldsymbol{FoPCe}_{\mathcal{D}}$ [81]

In order to use the classical sequent calculus **LK** [50], a pragmatic approach of 'separating the concern of validity from that of well-definedness' [81] can be adopted. Therefore, when proving a sequent $\mathbf{H} \vdash G$, two sequents need to be proved:

$$\boxed{\text{WD}_{\mathcal{D}}: \quad \vdash_{\mathcal{D}} \mathcal{D}(\mathbf{H} \vdash G)} \qquad \boxed{\text{Validity}_{\mathcal{D}}: \quad \mathbf{H} \vdash_{\mathcal{D}} G}$$

The $\text{WD}_{\mathcal{D}}$ proof obligation is equivalent to the original WD proof obligation since we know from (2.22) that '$\mathcal{D}(\mathcal{D}(H \vdash G)) \Leftrightarrow \top$'. To get $\text{Validity}_{\mathcal{D}}$, we add the extra hypothesis '$\mathcal{D}(H \vdash G)$' to Validity using the *cut* rule whose first antecedent can be discharged using the proof of WD [81].

The validity sequent '$\text{Validity}_{\mathcal{D}}$' is shown, in [81], to be equivalent to:

$$\widehat{\mathcal{D}}(\mathbf{H}), \mathcal{D}(G), \mathbf{H} \vdash G$$

where the $\widehat{\mathcal{D}}$ operator is the $\mathcal{D}$ operator extended for a finite set of formulae. This equivalence asserts that when proving the validity of a well-defined sequent, its hypotheses and goal can be assumed to be individually well-defined.

2.4.4.2 WD-Preserving Inference Rules

An inference rule is said to preserve well-definedness iff its consequent and antecedents are all well-defined sequents. Figure 2.10 introduces the theory $\boldsymbol{FoPCe}_{\mathcal{D}}$ (a collection of WD-preserving inference rules) as developed in [81, 82]. The well-definedness preserving proof rules are developed with a detour through the classical calculus, shown in Figure

2.9, and using the following (bridging) inference rule:

$$\frac{\widehat{\mathcal{D}}(H), \mathcal{D}(G), H \vdash G}{H \vdash_{\mathcal{D}} G} \vdash_{\mathcal{D}} eqv$$

The double inference line means that the rule can be used in both directions. As such, the bridging rule allows the passage between the classical and the well-definedness preserving proof calculi, and vice versa.

Note the additional antecedents in the cases of $cut_{\mathcal{D}}$ and $\forall hyp_{\mathcal{D}}$ rules compared to their classical counterparts. This is necessary since both rules introduce a new formula ($cut_{\mathcal{D}}$) or a new term ($\forall hyp_{\mathcal{D}}$), that may not be well-defined, into the proof. Note that other inference rules concerning derived logical operators can be derived using the inference rules of $\boldsymbol{FoPCe}_{\mathcal{D}}$. Note the use of the non-freeness constraint (x $\underline{\text{nfin}}$ H denoting 'x is not free in H'), defined in the usual way, in the universal quantification introduction rule $\forall goal_{\mathcal{D}}$.

The following two proof rules can be derived with a detour through $\vdash$ sequents (classical reasoning) [82]:

$$\frac{P, \mathcal{D}(Q) \ \vdash_{\mathcal{D}} \ Q}{P \ \vdash_{\mathcal{D}} \ Q} \ goal_{WD}$$

and

$$\frac{P, \mathcal{D}(P) \ \vdash_{\mathcal{D}} \ Q}{P \ \vdash_{\mathcal{D}} \ Q} \ hyp_{WD}$$

To give the reader an intuition into how rules are derived, we show how to derive the following rule:

$$\frac{}{H \vdash_{\mathcal{D}} \top} \top goal_{\mathcal{D}}$$

by means of the following proof tree:

$$\dfrac{\dfrac{\dfrac{}{\widehat{\mathcal{D}}(H), \mathcal{D}(\top), H, \bot \vdash \bot} \bot hyp}{\widehat{\mathcal{D}}(H), \mathcal{D}(\top), H \vdash \top} contr}{H \vdash_{\mathcal{D}} \top} \vdash_{\mathcal{D}} eqv$$

The following proof tree shows how $goal_{WD}$ is derived:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{D}(P), \mathcal{D}(Q), P \vdash \top} \top goal_{\mathcal{D}}}{\mathcal{D}(P), \mathcal{D}(Q), P \vdash \mathcal{D}(\mathcal{D}(Q))} 2.22 \quad \dfrac{P, \mathcal{D}(Q) \vdash_{\mathcal{D}} Q}{\mathcal{D}(P), \mathcal{D}(Q), P, \mathcal{D}(\mathcal{D}(Q)) \vdash Q} \vdash_{\mathcal{D}} eqv}{\mathcal{D}(P), \mathcal{D}(Q), P \vdash Q} cut}{P \ \vdash_{\mathcal{D}} \ Q} \vdash_{\mathcal{D}} eqv$$

For the remainder of this thesis, we may also use the well-definedness preserving proof rules shown in Figure 2.11.

$$\frac{}{H \vdash_{\mathcal{D}} \top}\ \top goal_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} P}{H \vdash_{\mathcal{D}} P \lor Q}\ \lor goal1_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} P \lor Q}\ \lor goal2_{\mathcal{D}}$$

$$\frac{H, P \vdash_{\mathcal{D}} R \quad H, Q \vdash_{\mathcal{D}} R}{H, P \lor Q \vdash_{\mathcal{D}} R}\ \lor hyp_{\mathcal{D}} \qquad \frac{H, P \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} P \Rightarrow Q}\ \Rightarrow goal_{\mathcal{D}}$$

$$\frac{H \vdash_{\mathcal{D}} P \quad H, Q \vdash_{\mathcal{D}} R}{H, P \Rightarrow Q \vdash_{\mathcal{D}} R}\ \Rightarrow hyp_{\mathcal{D}} \qquad \frac{H \vdash_{\mathcal{D}} P \Rightarrow Q \quad H \vdash_{\mathcal{D}} Q \Rightarrow P}{H \vdash_{\mathcal{D}} P \Leftrightarrow Q}\ \Leftrightarrow goal_{\mathcal{D}}$$

$$\frac{H, P \Rightarrow Q \vdash_{\mathcal{D}} R \quad H, Q \Rightarrow P \vdash_{\mathcal{D}} R}{H, P \Leftrightarrow Q \vdash_{\mathcal{D}} R}\ \Leftrightarrow hyp_{\mathcal{D}}$$

$$\frac{H \vdash_{\mathcal{D}} \mathcal{D}(E) \quad H \vdash_{\mathcal{D}} [x := E]P}{H \vdash_{\mathcal{D}} \exists x \cdot P}\ \exists goal_{\mathcal{D}} \qquad \frac{H, P \vdash_{\mathcal{D}} Q}{H, \exists x \cdot P \vdash_{\mathcal{D}} Q}\ \exists hyp_{\mathcal{D}}(x \ \underline{\text{nfin}}\ H \cup \{Q\})$$

$$\frac{P, \mathcal{D}(Q) \ \vdash_{\mathcal{D}}\ Q}{P \ \vdash_{\mathcal{D}}\ Q}\ goal_{WD} \qquad \frac{P, \mathcal{D}(P) \ \vdash_{\mathcal{D}}\ Q}{P \ \vdash_{\mathcal{D}}\ Q}\ hyp_{WD}$$

Figure 2.11: Additional Well-Definedness Preserving Inference Rules [81, 82, 102]

2.4.5 Proofs in Event-B

As mentioned in §2.4.4, when proving a sequent $\mathbf{H} \vdash G$ in Event-B, two sequents need to be proved:

$$\boxed{\text{WD}_{\mathcal{D}}: \quad \vdash_{\mathcal{D}} \mathcal{D}(\mathbf{H} \vdash G)} \qquad \boxed{\text{Validity}_{\mathcal{D}}: \quad \mathbf{H} \vdash_{\mathcal{D}} G}$$

The WD-preserving proof calculus ($\boldsymbol{FoPCe}_{\mathcal{D}}$) can be used to prove both sequents:

- '$\text{WD}_{\mathcal{D}}$' for each proof obligation are factored out by proving that the source models (i.e., from which the proof obligations are generated) are well-defined [25]. Proof obligations generated from well-defined models are guaranteed to be well-defined. This reduces the number of proofs that need to be carried out [81].
- '$\text{Validity}_{\mathcal{D}}$' for each proof obligation can be discharged using the WD-preserving proof calculus.

Example. Assuming a suitable theory of arithmetics, consider the case where the modeller specifies the following theorem in a context:

$$\forall x : \mathbb{Z} \cdot x \div x = 1 \tag{2.28}$$

Two proof obligations are generated to establish the validity of (2.28):

$$(\text{WD}_{\mathcal{D}}) \qquad \vdash_{\mathcal{D}} \forall x : \mathbb{Z} \cdot x \neq 0 \tag{2.29}$$

$$(\text{Validity}_{\mathcal{D}}) \qquad \vdash_{\mathcal{D}} \forall x : \mathbb{Z} \cdot x \div x = 1 \tag{2.30}$$

The Sequent (2.29) cannot be discharged since its negation is provable and the calculus (***FoPCe***$_{\mathcal{D}}$) is shown to be sound in [102]. The negation of Sequent (2.29) is the following sequent,

$$\vdash_{\mathcal{D}} \exists x : \mathbb{Z} \cdot x = 0$$

which is shown to be provable by means of the following proof tree:

$$\dfrac{\dfrac{}{\vdash_{\mathcal{D}} \top}\ \top goal_{\mathcal{D}} \quad \dfrac{}{\vdash_{\mathcal{D}} 0 = 0}\ = goal_{\mathcal{D}}}{\vdash_{\mathcal{D}} \exists x : \mathbb{Z} \cdot x = 0}\ \exists goal_{\mathcal{D}}$$

However, the Sequent (2.30) can be discharged since we have the following proof tree:

$$\dfrac{\dfrac{}{\forall x : \mathbb{Z} \cdot x \neq 0 \vdash_{\mathcal{D}} \top}\ \top goal_{\mathcal{D}} \quad \dfrac{\dfrac{}{\bot \vdash_{\mathcal{D}} \forall x : \mathbb{Z} \cdot x \div x = 1}\ \bot hyp_{\mathcal{D}}}{0 \neq 0 \vdash_{\mathcal{D}} \forall x : \mathbb{Z} \cdot x \div x = 1}\ \forall hyp_{\mathcal{D}}}{\dfrac{\forall x : \mathbb{Z} \cdot x \neq 0 \vdash_{\mathcal{D}} \forall x : \mathbb{Z} \cdot x \div x = 1}{\vdash_{\mathcal{D}} \forall x : \mathbb{Z} \cdot x \div x = 1}\ goal_{WD}}$$

In summary, theorem (2.28) can be shown to be valid but not well-defined using the well-definedness preserving calculus (***FoPCe***$_{\mathcal{D}}$).

Summary. In this section, we presented an overview of the proof calculus used to reason in Event-B. We have shown how partial functions are added to a theory by means of a conditional definition in §2.4.2. Moreover, we introduced the well-definedness operator that generates well-definedness conditions for terms and formulae. We also presented the work of Mehta [81, 110] regarding the well-definedness preserving proof calculus (***FoPCe***$_{\mathcal{D}}$). We concluded this section by briefly discussing proofs in Event-B by means of a simple example.

2.5 Other Formalisms

In this section, three formalisms are introduced: Isabelle/HOL [90, 94, 95], VDM [28, 70] and PVS [91, 59]. The aim of this section is to highlight major differences between Event-B and other established methodologies, and to investigate how these formalisms can influence our approach to achieve the objectives outlined in §1.2. In the following three subsections (§2.5.1, §2.5.3 and §2.5.2), we briefly describe Isabelle/HOL, VDM and PVS. In §2.5.4, advanced features of the aforementioned formalisms will be discussed. The

choice of Isabelle/HOL, PVS and VDM is taken because these three formal techniques are known as powerful modelling tools that have been used in non-trivial applications. Furthermore, VDM uses the logic of partial functions that deals with ill-definedness. PVS adopts a simpler approach to ill-definedness by generating type correctness conditions (TCC's) [91, 59]. Finally, Isabelle is an established theorem prover that has been used to formalise many logics including a shallow embedding for Event-B [101].

2.5.1 Isabelle/HOL

Isabelle is a generic theorem prover developed by Paulson [93]. Isabelle is generic in the sense that it offers a meta-logic in which many object logics can be formalised. The meta-logic of Isabelle is intuitionistic higher-order logic with implication, universal quantifiers and equality. Isabelle has been referred to as the *next 700 provers* [93].

Isabelle borrows many ideas from the earlier LCF (Logic of Computable Functions) theorem prover developed by Milner [85]. The meta-language Standard ML [88] is used to manipulate formulae. Theorems in the LCF system are propositions of a special "theorem" abstract datatype. The ML type system ensures that theorems can only be derived using the inference rules specified by the operations of the abstract type. Proofs are carried out by means of tactics and tacticals written as functions in ML. LCF represents the backward inference rule

$$\frac{A \quad B}{A \wedge B} \wedge i$$

as a function that maps theorems A and B to the new theorem $A \wedge B$. In Isabelle, however, the meta-logic is used to express such a rule as follows [93]:

$$\bigwedge A \cdot \bigwedge B \cdot \ [\![A]\!] \Rightarrow \ ([\![B]\!] \Rightarrow \ [\![A \wedge B]\!]) \ .$$

The brackets $[\![\ ]\!]$ are used to enclose object-logic formulae, whereas meta-logic formulae reside outside the brackets. Effectively, Isabelle/HOL is the Isabelle theorem prover instantiation for higher-order logic. Note that, in Event-B, the programming language Java is used to specify proof rules; as such it could be considered as a meta-language for Event-B in the same way Standard ML is considered as a meta-language for LCF [85].

2.5.1.1 The Language

The specification language of Isabelle is inspired by functional programming languages. A theory is a component that may contain Isabelle declarations, definitions and proofs. The module system in Isabelle allows the importing of multiple theories. A theory in Isabelle may define types, terms and formulae. The types found in theories are: (1) base

types, e.g., *bool*, (2) type variables, e.g., *'a*, (3) function types, and (4) type constructors, e.g., *'a list*. Terms are formed by applying functions to arguments. Note that, in Isabelle, functions are total, and can be declared polymorphically [90].

Isabelle allows the definition of axiomatic type classes [108]. In a type class, polymorphic declarations for functions are given. Moreover, additional properties of these functions can be stated, and these can be used as axioms in the rest of the theory. The modeller can instantiate type classes by providing appropriate bodies for the functions, and proving that the properties hold. Overloading, in Isabelle, is only allowed in the case of polymorphic functions with a single polymorphic type [59].

Inductive and co-inductive datatypes can be defined using Isabelle. Support for primitive recursive functions is available. Furthermore, well-founded recursive functions can be defined together with a measure function to show their termination [59]. Conveniently, Isabelle automatically generates induction principles for each user-defined recursive datatype.

The syntax of Isabelle can easily be extended. The tool provides the user with the facility to define infix and mixfix operators. The user can also specify priorities and preferred syntax for new operators. For example, **[1,2]** can be made to represent the cumbersome **cons 1 (cons 2 nil)**. This is particularly crucial for Isabelle given that it was conceived to be a generic theorem prover [93].

2.5.1.2 The Prover

Goals in Isabelle have the form $[A_1; ...; A_n] \Rightarrow B$ where A_i is the list of assumptions and B is the conclusion. Resolution with higher-order unification is the main proof method in Isabelle. Resolution works on the goal's assumption, generating new assumptions. Resolution yields both backward and forward proofs. Backward proof works by unifying a goal with the conclusion of a rule, whose premises, then, become the new sub-goals. Forward proof works by unifying theorems (or assumptions) with the premise of a rule, deriving a new theorem (or assumption) [95, 93].

A tactic, in Isabelle, transforms a proof goal into several sub-goals, and provides a justification for the proof step. Isabelle is geared for backward proof by providing a large collection of useful tactics [59]. An important mechanism for the working of tactics is the instantiation of unknowns and variables in goals and assumptions. As the instantiation mechanism may provide a number of instantiations, instantiations are tried one after the other until one instantiation is satisfactory. An important component, in this process, is the backtracking procedure, that is called upon in case an instantiation is not satisfactory [59].

Tactics in Isabelle can be classified into several categories [59]:

- Basic tactics: this includes resolution, **RS**, and **assume_tac**. Resolution works by unifying the conclusion of a theorem with the conclusion of the goal. If the unification succeeds, a suitable substitution is provided. The resolution method, then, creates a new set of sub-goals corresponding to the assumptions of the theorem after applying the provided substitution. The basic tactic **assume_tac** works by unifying the conclusion of the goal with one of its assumptions.

- Induction: the tactic *induct_tac* does resolution with an appropriate induction rule.

- Simplification: this uses tactics for rewriting. For every created theory, a simplification set can be built from theorems, axioms and definitions. The simplification set can be used to rewrite a goal. Note that the Isabelle prover employs a special strategy to deal with permutative rewrite rules, i.e., rewrites whose sides are equal up to renaming of variables. A lexical order is observed, and a permutative rewrite rule can only be applied if it decreases the term with respect to the defined lexical order.

- Classical reasoning: an example is **blast_tac** which uses a tableau prover coded in ML [59].

- Bureaucratic tactics: an example is **rotate_tac** which can be used to change the order of assumptions. Changing the order of assumptions may be necessary for rewriting with a particular assumption.

Isabelle has a powerful tactical language. A tactical is a function that creates complex tactics using the basic ones. The tactical *then* groups together two tactics and applies them sequentially to the goal. The tactical language in Isabelle is Standard ML.

Example. The specification of a sequence is defined in the following theory. `Datatype` and `FunDef` are the imported theories.

```
theory Sequence
imports Datatype FunDef
begin
```

The following line will create a sequence datatype using an inductive definition. When it is analysed by Isabelle, some properties will be readily available regarding the datatype itself.

```
datatype 'a sequence = Nil ("[]")
    | Cons 'a"'a sequence" (infixr "#" 65)
```

Next, functions such as head (returns the topmost element of the sequence), size and tail are defined. Note, in particular, the reliance of Isabelle/HOL on pattern matching; a feature inherited from its implementation language ML. In particular, the size function

is defined recursively.

```
fun head ::  "'a sequence ⇒ 'a set" where
     "head [] = {}" | "head (x#xs) = {x}"
primrec size ::  "'a sequence ⇒ nat" where
    "size [] = 0"|"size (x#xs) = size xs + 1"
fun tail ::  "'a sequence ⇒ 'a sequence" where
    "tail [] = []"|"tail (x#xs) = xs"
```

The following lemma formalises the logical relationship between the size and tail functions. Its proof is straightforward. It inducts on the sequence `xs` using the tactic `induct_tac`. Using the `auto` tactic completes the proof.

```
lemma tail_size_rel:  "size (tail (x#xs)) = size(xs)"
  apply(induct_tac xs)
  apply(auto)
  done
```

The following function is another way of describing the second constructor of the sequence datatype.

```
fun add ::  "'a sequence ⇒ 'a ⇒'a sequence" where
    "add xs a = a#xs"
```

The append function is defined in a recursive fashion below.

```
primrec append ::  "'a sequence ⇒ 'a sequence ⇒ 'a sequence" where
    "append [] xs = xs"|"append (x#xs) ys = x#(append xs ys)"
```

A theorem relating the append and size functions is stated and defined.

```
theorem append_size_rel:  "size(append xs ys) = size xs + size ys"
   apply(induct_tac xs)
    apply(auto)
    done
```

The interesting map function is defined and theorems relating it to other functions are stated and proved.

```
primrec map ::  "'a sequence ⇒ ('a ⇒ 'b) ⇒ 'b sequence" where
    "map [] f = []"|"map (x#xs) f = (f x)#(map xs f)"
theorem map_size_preservation:  "size (map xs f) = size xs"
  apply(induct_tac xs)
  apply(auto)
  done
```

```
theorem tail_map_rel:  "map (tail xs) f = tail (map xs f)"
  apply(induct xs)
  apply(auto)
  done
```

2.5.2 PVS

The Prototype Verification System [92, 91] was developed by SRI International Computer Science Laboratory. Work on PVS started in 1990, and the first version was released in 1993. PVS is written in the Lisp programming language, and is integrated with the Emacs editor. Unlike Isabelle, PVS source code is not freely available.

PVS employs classical typed higher-order logic, extended with predicate subtypes and dependent types [59]. PVS defines a number of built-in types including booleans, lists, integers and reals. The usual operations on these types are hardcoded in PVS. Types can also be constructed using type constructors, e.g., function types, product types, records and recursive datatypes.

A predicate subtype, in PVS, is a type constructed by collecting elements of a particular type that satisfy a given predicate. A notable example is the set of non-zero reals. The set of non-zero reals is used to define the division operator. The authors of [59] argue that the use of predicate subtypes improves the readability of specifications, and helps with detecting semantic errors related to them.

In PVS, dependent types can be constructed using predicate subtypes. In [59], the following example is provided:

```
Ex_Array[T: TYPE]: THEORY
BEGIN
Ex_Array: TYPE = [# length: nat, val: [below(length) -> T] #]
END Ex_Array
```

In this example, `Ex_Array` is a record type with two fields. The first field (`length`) denotes the length of the array. The second field (`val`) is the array of values stored at each index. The domain of `val` is the predicate subtype `below(length)` of the natural numbers less than `length`. As such, the type of `val` depends on its length.

2.5.2.1 The Language

PVS provides an integrated environment to create and reason about formal specifications. The specification language employed by PVS is based on higher order logic. It

has a strong type system that incorporates a rich built-in set of types, type constructors, and predicate subtypes. Definitions and axioms are the building blocks of specifications which are organised into theories and datatypes [92].

A specification written in PVS is made of theories. Each theory exhibits a signature that describes the different types and constants it uses. It also contains the definitions, axioms and theorems that govern the signature. A theory can be based on other theories, for instance, a stack theory can be modelled by means of a sequence theory.

Another important characteristic of PVS theories is parametrisation. A specification in PVS is usually divided into several theories, and each theory can be parametrised on types and values. In the example of stacks, the defining theory can be parametric on the type of the elements it stores. A theory can be imported, and all its parameters have to be instantiated by the importing theory. The assuming clause in PVS can be used to constrain its parameters. When a theory with an assuming clause is imported, type correctness conditions (TCC's) are generated to ensure that the assumptions of the imported theory hold for the parameter instantiations.

Polymorphism is not available in PVS. However, it can be approximated by the use of type parameters to parametrise theories. A polymorphic function can be defined in a theory parametrised by the type variables of the said function. As pointed out in [59], this approach may not always be convenient, because when a theory is imported all its parameters must have a value, regardless of whether they are used by the required function.

PVS allows operator overloading. This means that functions within the same theory may have the same name as long as they differ in their types. Different theories can define functions of the same name, even if they have the same type. The name of the theory can be used as prefix to distinguish similarly named functions [91, 59].

Inductive datatypes and recursive functions can be defined in PVS. An induction principle and a number of standard functions such as map and reduce are automatically generated by the tool. All functions in PVS must be total, and as such, recursive functions must be shown to terminate by providing a measure function, Type correctness conditions are generated to ensure the measure function decreases with every recursive call [92, 91, 59].

PVS has a much fixed syntax. The standard operators on the sets of reals, integers and booleans are built-in to the language. As noted in [59], PVS, sometimes, uses uncommon syntax for common operators, e.g., `[A,B]` for the Cartesian product of `A` and `B`.

2.5.2.2 The Prover

The sequent calculus is used to represent goals in PVS. $A_1, .., A_n \vdash B_1, ..., B_m$ is a sequent where A_i are the hypotheses and B_j are the conclusions. This is equivalent to $A_1 \wedge ... \wedge A_n \Rightarrow B_1 \vee ... \vee B_m$. The proof commands in PVS can be categorised into [59]:

- Creative proof commands: Examples of these commands include: **induct** to initiate a proof by induction, **inst** to instantiate a quantified predicate and **case** to make a case distinction.
- Bureaucratic proof commands: Examples include **flatten** for disjunctive simplification, **expand** to expand definitions, and **hide** to hide assumptions which have become irrelevant.
- Powerful proof commands: These are intended to discharge trivial goals. Examples include **simplify** for simplification. A more powerful command is **assert** which uses the simplification command as well as the available decision procedure, e.g., arithmetic decision procedures.

PVS has a limited tactical language that includes sequencing, backtracking, branching and recursion. Other proof strategies can be implemented in Lisp [59].

Example. PVS has a powerful datatype mechanism. In this example, we show how the sequence structure can be defined in PVS. The following snippet will create the sequence datatype with two constructors. Note that the datatype is parameterised on type `S`.

```
Seq[S:TYPE] : DATATYPE
BEGIN
   Nil:  Nil?
   Cons(head:S, tail:  Seq):NonNil?
END Seq
```

This will automatically create a theory that underlies the sequence datatype (saved in a file Seq_adt.pvs). Here are some extracts from the resulting theory.

```
Seq_adt[S: TYPE]: THEORY
 BEGIN
  Seq: TYPE
  Nil?, NonNil?: [Seq -> boolean]
  Nil: (Nil?)
  Cons: [[S, Seq] -> (NonNil?)]
  head: [(NonNil?) -> S]
  tail: [(NonNil?) -> Seq]
```

In the previous snippet, a sequence type as well as two subtypes `(Nil?)` and `(NonNil?)` are declared. Furthermore, functions `head`, `tail` and `Cons` are defined by giving their types. The following axiom defines the induction mechanism of sequences which is expressible by quantifying over predicates.

```
Seq_induction: AXIOM
    FORALL (p: [Seq -> boolean]):
      (p(Nil) AND
        (FORALL (Cons1_var: S, Cons2_var: Seq):
           p(Cons2_var) IMPLIES p(Cons(Cons1_var, Cons2_var))))
       IMPLIES (FORALL (Seq_var: Seq): p(Seq_var));
```

After defining the sequence datatype, we can import the resulting theory and use it for modeling.

```
Sequence [S: TYPE]: THEORY
BEGIN
importing Seq_adt[S]
s, s1: VAR Seq
e:VAR S
n:VAR nat
```

The size functions is defined in terms of the function `reduce_nat`. This functions along many others were generated when the datatype is created.

```
size(s): nat = reduce_nat(0, lambda e,n :1+n)(s)
```

Two theorems relating `head, tail` and `size` are stated. Their proof is carried out using a powerful PVS tactic called **induct-and-rewrite!**.

```
head_tail_rel: THEOREM
       NonNil?(s) => Cons(head(s), tail(s)) = s
size_tail_rel: THEOREM
       NonNil?(s) => size(s) = size(tail(s))+1
```

Finally, the appending function is recursively defined, and a theorem regarding its relationship with the size function is stated. The proof of the theorem is achieved by using the tactic **induct-and-rewrite**.

```
append(s, s1): recursive Seq =
  (if Nil?(s) then s1 else
```

```
   Cons(head(s), append(tail(s), s1)) endif)
  measure (lambda s,s1:size(s)+size(s1))
append_size_rel: THEOREM
 size(append(s,s1)) = size(s)+size(s1)
END Sequence
```

2.5.3 VDM

The Vienna Development Method (VDM) [28, 70] is a system modelling and development method much like Event-B in its general objectives. It provides rules to verify the different steps of system development including data reification and operation decomposition. In this subsection, we will present a brief overview of VDM with a focus on its specification language and the underlying logic.

LPF (Logic of Partial Functions) [33, 24, 32, 67] is used for reasoning about VDM models. LPF is a three-valued first-order predicate logic designed for reasoning about languages with partial functions. LPF gives non-classical interpretations to the logical connectives and quantifiers. Atomic formulae that contain non-denoting terms may be logically neither true nor false. The logical connectives and quantifiers are augmented in order to handle operands that are neither true nor false. However, the classical truth and falsehood conditions are retained as much as possible in order to minimise the deviation from intuitive interpretations.

A typed version of LPF was introduced by Jones and Middelburg [68], and is the logic used for reasoning about VDM models. In addition to the logical values **true** and **false**, LPF admits undefined (also called non-denoting) terms, and uses the value $\perp_{\mathbb{B}}$ to account for such terms. The truth tables for negation (see Figure 2.12) and disjunction (see Figure 2.13) may be thought of as describing a 'parallel lazy evaluation of the operands' [67].

	$\neg$
true	**false**
false	**true**
$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$

Figure 2.12: Truth Table for $\neg$ in LPF

An important feature of LPF is the absence of the law of excluded middle:

$$\frac{}{e \vee \neg e}\ \boxed{\text{Excl-Mid}}$$

$\vee$	**true**	**false**	$\perp_{\mathbb{B}}$
true	**true**	**true**	**true**
false	**true**	**false**	$\perp_{\mathbb{B}}$
$\perp_{\mathbb{B}}$	**true**	$\perp_{\mathbb{B}}$	$\perp_{\mathbb{B}}$

Figure 2.13: Truth Table for $\vee$ in LPF

As a consequence, classical deduction cannot be used:

$$\frac{e_1 \vdash e_2}{e_1 \Rightarrow e_2}\ \boxed{\text{Deduction}}$$

The well-definedness operator δ is used in LPF to recover the power of two-valued classical logic, and is defined as follows:

$$\delta e \ \widehat{=}\ e \vee \neg e\ .$$

The deduction rule can, then, be rewritten to the valid rule:

$$\frac{\delta e_1 \quad e_1 \vdash e_2}{e_1 \Rightarrow e_2}\ \boxed{\Rightarrow \mathsf{I}}$$

2.5.3.1 The Language

VDM-SL (shorthand for VDM Specification Language) is used to create specifications in a model-oriented approach. The data model of a specification written in VDM-SL defines: (1) the abstraction of the data types that are needed by the system, (2) the collection of operations that describe the required behaviour of the system in question. In some cases, the system may be required to posses a state in which case a state type is defined as part of the data model. The operations describing the behaviour of a system define a relation between input and output values of defined types. In the presence of system state, the operations may change the state as a side effect of maintaining the relation between their input and output.

Abstraction is an important technique to address the complexity of systems. The data model defined in a specification is an abstraction of the various data types that will appear in the final implementation. Data reification techniques allow the specification to evolve in a way that makes its data model more closely approximate the data types of the implementation. Each refinement step is shown to maintain the requirements of the more abstract specifications.

Mathematical structures, such as relations, are used to specify model operations. In the final implementation, however, operations are turned into executable programs. Operation decomposition techniques facilitate the introduction of useful programming constructs in the definition of operations as the specification progressively evolves into a

concrete implementation. VDM-SL provides a collection of predefined operation combinators that enable imperative-style operations specifications [28].

2.5.3.2 The Prover

A collection of tools are available for developing models in VDM. This includes: VDM-Tools [2] and Overture [72]. However, currently there is no VDM-specific theorem prover. An initiative to enable automated proof support for VDM is under way as part of the Overture community project [106]. A semantic-preserving translator has been created; it works by porting VDM proof obligations to be discharged by HOL [57] theorem prover.

2.5.4 A Comparison: Event-B, Isabelle/HOL, PVS and VDM

In this section, we provide a summary of the differences between Event-B, Isabelle/HOL, PVS and VDM. In the comparison that follows, we consider the Event-B methodology prior to our work. To enhance readability, the comparison is provided in tabular format (see Table 2.1, 2.2 and 2.3), and is divided into three parts: the logic, the specification language, and the prover. The following key points summarise the criteria against which the three formalisms will be compared:

- Logic: the formalisms are compared with respect to the logic used, its expressiveness and how it handles partiality.
- Specification Language: the three formalisms are contrasted with regards to the usability, expressiveness and extensibility of the specification language.
- Prover: the formalisms are compared in terms of prover effectiveness (i.e., how powerful is the support for automatic proofs), extensibility and soundness.

Note that VDM relies on external provers to discharge proof obligations, as such, VDM is not considered for the comparison of provers. In the comparison tables below, the use 'N/A' signifies that the feature is not supported, or that we cannot make a judgement based on the available documentation.

2.5.5 A Reflection

In this section, we provided an overview of three widely used formalisms. We, briefly, described their logics, specification languages and provers. In the context of the comparison in §2.5.4, we single out the following aspects of Event-B that we aim to improve:

	Event-B	Isabelle/HOL	PVS	VDM
The Logic	Set theory	Typed HOL	Typed HOL	LPF
Predicate Subtypes	N/A	N/A	++	N/A
Dependent Types	N/A	N/A	++	N/A
Polymorphism	N/A	++	-	+
Abstract Datatypes	N/A	++/+	++/+	N/A
Recursive Functions	N/A	++/+	++/+	-

Table 2.1: Comparison of Logics

	Event-B	Isabelle/HOL	PVS	VDM
Flexible Syntax	-	++	-	-
Module System	-	+	++/+	+
Overloading	N/A	-	++	-
Libraries	N/A	+	++/+	+

Table 2.2: Comparison of Specification Languages

	Event-B	Isabelle/HOL	PVS
Automation	+	+	+
Proof Management	++	+/-	++
Tactical Language	-	++	-
Arithmetics	-	+/-	++
Soundness	-	++	-

Table 2.3: Comparison of Provers

1. Support for polymorphism: Event-B does not support user-defined polymorphic operators. In this thesis, we show how user can contribute polymorphic operators in a sound and usable way. We will show in Chapter 4 that our approach to address this particular issue resembles the approach taken by Isabelle/HOL rather than that taken by PVS.

2. Abstract datatypes and recursive functions: a minor contribution of this thesis (§4.8) is the provision of a mechanism to specify inductive datatypes and recursive operators. The contribution in §4.8 is strongly influence by datatypes in Isabelle/HOL.

3. Syntax flexibility: in §4.7, we will show how to ensure that new syntax can be contributed to the Event-B mathematical language without compromising the soundness of the formalism.

4. Proof management and soundness: in Chapter 4, we will show how the Event-B prover can be augmented with new proof rules in a usable and sound fashion. The

work in Chapter 3 was motivated by the presence of unsound rewriting rules in Rodin.

5. Module system: in §4.2, we show how to structure Event-B models in a way that promotes reusability.

Chapter 5 describes the practical contribution by which we address the aforementioned concerns. Chapter 6 showcases the use of the Theory plug-in by means of a few relatively simple examples.

2.6 The Logic of Event-B

Schmalz defines the Event-B logic using a shallow embedding in Isabelle/HOL [90].[9] A deep embedding of a logic in Isabelle requires (1) defining the syntax of the object logic as a datatype, (2) providing semantics of the object logic, and (3) proving that the axioms governing the syntax are sound with respect to the semantics. A shallow embedding does not requires steps (1) and (2) [52]. As a result, shallow embedding can be thought of as a syntactic translation.

Schmalz provides a comprehensive specification of the logic of Event-B in one document [90]. He gives semantics, devises soundness preserving extension methods, develops a proof calculus similar to [81], and proves its soundness. [90] presents a formal language for expressing rules (including non-freeness conditions) and show how to reason in Event-B about the soundness of rules.

The Event-B logic has a Hindler-Milner style type system [102] similar to Isabelle/HOL and ML [88]. Type operators such as $\times$ and $\leftrightarrow$ are defined by means of their Isabelle/HOL counterparts.[10] Type substitutions are central to a logic that supports polymorphism, and are also introduced. Binders, terms and formulae are introduced are assigned Isabelle/HOL semantics by means of a number of higher-order logic constructs. Note that Schmalz considers formulae (i.e., predicates) to have a boolean type $\mathcal{B}$. Ways of conservatively extending the Event-B logic are outlined (see Chapter 5 of [102]).

The proof system of Event-B is shown to be sound in [102]. We say that a rule is sound if it is derived from the basic rules of the well-definedness preserving proof calculus with or without detour through the classical proof calculus. For instance, we have shown in §2.4.4.2 that the following two rules are sound:

$$\frac{P, \mathcal{D}(Q) \vdash_{\mathcal{D}} Q}{P \vdash_{\mathcal{D}} Q}\ goal_{WD}$$

[9] A deep embedding requires Event-B logic to be defined as an object logic in Isabelle. This is a highly involved process, and may render useful proof procedure of Isabelle/HOL unusable.

[10] $\mathbb{Z}$ is considered a type operator with a zero arity.

$$\frac{}{H \vdash_{\mathcal{D}} \top} \top goal_{\mathcal{D}}$$

with detour through the classical proof calculus, i.e., $\vdash$ sequents. More generally, we say that a proof extension, i.e., a rewrite or an inference rule, or a polymorphic theorem, is sound if its application can be justified by a proof construction using the rules of the well-definedness preserving proof calculus. In the case of language extensions, soundness of an operator definition requires the satisfaction of the conditions stipulated in in [102] regarding conservative extensions.

In Chapter 3, we will only use an untyped fragment of the Event-B logic. The work in Chapter 3 can be considered as a complement to Mehta's work in [81]. Suitably, it was decided to use a similar fragment of the logic of Event-B. However, for the work on language extensibility (polymorphic operators in particular), we base our discussion and justify our development using the results presented by Schmalz in [102] regarding conservative extensions.

2.7 Summary

In this chapter, we presented an overview of the different concepts that will come into play in subsequent chapters. Firstly, formal methods were introduced to provide the general context for this work. Next, the focus was placed on Event-B and the Rodin platform which provides the practical setting for this thesis. The proof infrastructure of Rodin was presented and its shortcomings identified. The proof system used in Event-B is described. Next, a brief comparison was carried out between Event-B and three other formalisms. Finally, we presented a brief overview of the logic of Event-B as described in [102]. In the next chapter, we explore the integration of rewriting into the well-definedness preserving proof system using an untyped fragment of the Event-B syntax.

Chapter 3

Rewriting and Well-Definedness within a Proof System

In this chapter, we provide a unifying study of term rewriting systems and the important notion of well-definedness. The sequent calculus used in Event-B reasoning takes into consideration partiality and its implications. The Event-B proof system is described in details by Mehta [82, 81] and Schmalz [102]. Our aim is to show how rewriting preserves equality/equivalence and well-definedness of terms and formulae in the logic defined in §2.4. Important properties regarding well-definedness will be examined, and the conditions under which rewriting can be performed in a sound way are singled out. The results appearing in this chapter have been published in [77].

Rewriting is an important component of theorem proving. All major theorem provers have mechanisms for incorporating rewriting with the employed proof system. Event-B employs a well-definedness preserving proof system that is described by Mehta in his thesis [82]. We present an approach that facilitates the integration of rewriting into such a calculus. Our approach to rewriting aims to combine two important features: (1) show how well-definedness can be preserved when rewriting, (2) provide a simple way to apply rewrite rules.

This chapter is structured in the following way. We begin by presenting important concepts of rewriting including positions and substitutions. We then show how well-definedness propagates through positions and across substitutions. Next, the sufficient conditions under which a rewrite rule preserves well-definedness are singled out, and the results are summed up succinctly in Theorem 3.3. A large proof effort is carried out in this chapter, and the reader may be advised to skip proofs at a first pass. We also present two ways in which conditional rewrite rules can be used, and we show cases where this can be simplified. We conclude by describing related work.

3.1 Term Rewriting Systems

Term rewriting systems [19, 42, 96] are reduction systems where terms can be reduced to other terms by application of rewrite rules. Term rewriting systems play a major role in various disciplines including abstract datatype specifications, implementation of functional programming languages and automated reasoning. The λ-calculus [34], which is a term rewriting system, played a major role in mathematical logic. In this section, we briefly present some important term rewriting concepts. The notion of positions is central to term rewriting, and provides a mechanism to uniquely identify subterms (or subformulae). The important concept of a substitution will be introduced, and some interesting properties will be described.

3.1.1 Positions

The structure of terms and formulae can be effectively described using a tree. Using a standard numbering of nodes of the tree by strings of positive integers, it is straightforward to refer to positions in terms and formulae. For example, consider the formula $\varphi \wedge \psi$. The empty string ϵ identifies the root position, and refers to $\varphi \wedge \psi$. The position 1 refers to φ, whereas 2 refers to ψ. The following definition describes the concept of a position more formally.

Definition 3.1 (Position)**.** Let s be a term, and φ be a formula.

1. The set of **positions** of the term s is the set $\mathcal{P}os(s)$ of strings over the alphabet of positive integers, which is inductively defined as follows:
   - if $s = x \in V$, then $\mathcal{P}os(s) = \{\epsilon\}$, where ϵ denotes the empty string.
   - if $s = f(s_1, ..., s_n)$, then
   $$\mathcal{P}os(s) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{P}os(s_i)\}$$
2. The set of **positions** of the formula φ is the set $\mathcal{P}os(\varphi)$ of strings over the alphabet of positive integers, which is inductively defined as follows:
   - if φ is of the form $p(t_1, ..., t_n)$ where $p \in P$, then
   $$\mathcal{P}os(\varphi) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \mathcal{P}os(t_i)\}$$
   - if φ is of the form $t_1 = t_2$, then
   $$\mathcal{P}os(\varphi) = \{\epsilon\} \cup \bigcup_{i=1}^{2} \{ip \mid p \in \mathcal{P}os(t_i)\}$$

- if φ is of the form $\bot$, then

$$\mathcal{P}os(\varphi) = \{\epsilon\}$$

- if φ is of the form $\varphi_1 \wedge \varphi_2$, then

$$\mathcal{P}os(\varphi) = \{\epsilon\} \cup \bigcup_{i=1}^{2} \{ip \mid p \in \mathcal{P}os(\varphi_i)\}$$

- if φ is of the form $\neg\varphi_1$ or $\forall x \cdot \varphi_1$, then

$$\mathcal{P}os(\varphi) = \{\epsilon\} \cup \{1p \mid p \in \mathcal{P}os(\varphi_1)\}$$

We use the notation $s|_p$ where s is a term, to refer to the subterm of s at position $p \in \mathcal{P}os(s)$. Similarly, we use the notation $\varphi|_p$ to refer to the subterm or subformula of formula φ at position p. Moreover, the notation $s[t]_p$ refers to the term obtained by replacing the subterm $s|_p$ by t in s. Analogously, $\varphi[w]_p$ such that w and $\varphi|_p$ are both formulae or both terms, refers to the formula obtained from φ by replacing $\varphi|_p$ by w. For example, the notation $(\varphi_1 \wedge \varphi_2)|_1$ denotes the subformula φ_1. Similarly, the notation $(\varphi_1 \wedge \varphi_2)|_2$ denotes the subformula φ_2. Finally, for the rest of this chapter, we assume a syntactic operator

$$\mathcal{V}ar : (F_\Sigma \cup T_\Sigma) \to \mathbb{P}(V)$$

such that for a given term or formula t, $\mathcal{V}ar(t)$ is the set of its free variables.

3.1.2 Substitutions

In the language signature Σ defined in §2.4, a function with zero arity is called a constant[1]. One of the major differences between constants and variables is that a variable can be substituted for by a term. The following definition describes the notion of substitutions more formally.

Definition 3.2 (Substitution)**.** A T_Σ-**substitution**, or simply substitution if the set of terms is clear from the context, is a function $\sigma : V \to T_\Sigma$ such that $\sigma(x) \neq x$ for only finitely many variables x's. The finite set of variables that σ does not map to themselves is the **domain** of σ, i.e.,

$$\mathcal{D}om(\sigma) \;=\; \{x \in V \mid \sigma(x) \neq x\}$$

The **range** of a substitution σ is the set of terms which are the images of the variables in the domain of the substitution σ, and is formally defined as follows:

$$\mathcal{R}an(\sigma) \;=\; \{t \in T_\Sigma \mid \exists x \cdot x \in \mathcal{D}om(\sigma) \wedge t = \sigma(x)\}$$

[1]This is different from Event-B constants defined as part of contexts.

A substitution σ is said to instantiate variable x if $x \in \mathcal{D}om(\sigma)$. The application of a substitution σ to a term (or a formula) q *simultaneously* replaces occurrences of all variables in $\mathcal{V}ar(q) \cap \mathcal{D}om(\sigma)$ by their respective σ-images. A substitution σ can be extended to a mapping $\hat{\sigma} : T_\Sigma \to T_\Sigma$ such that:

$$\begin{aligned} \hat{\sigma}(x) &= \sigma(x) \text{ if } x \in V \\ \hat{\sigma}(f(s_1, ..., s_n)) &= f(\hat{\sigma}(s_1), ..., \hat{\sigma}(s_n)) \text{ if } f \in F \end{aligned}$$

The composition of two substitutions σ and τ is the substitution $\sigma\tau$ such that $\sigma\tau(x) = \hat{\sigma}(\tau(x))$. For the rest of this chapter, we use σ to also stand for $\hat{\sigma}$, and restrict substitutions according to the following definition:

Definition 3.3 (Idempotent Substitution)**.** A substitution σ is said to be idempotent if $\sigma = \sigma\sigma$.

The repetitive application of an idempotent substitution yields the same result as a single application. We have the following important corollary [19]:

Corollary 3.1. *A substitution σ is idempotent iff*

$$[\bigcup_{t \in \mathcal{R}an(\sigma)} \mathcal{V}ar(t)] \cap \mathcal{D}om(\sigma) = \varnothing$$

Corollary 3.1 states that the variables occurring in the terms of the substitution's range are completely independent from the variables of its domain. Intuitively, this means that an idempotent substitution can be *simulated* by a syntactic replacement as follows:

$$\sigma(l) \mathrel{\widehat{=}} [x_1 := \sigma(x_1)]...[x_n := \sigma(x_n)]l \tag{3.1}$$

such that l is a term and $x_1, ..., x_n$ are the free variables occurring in l. This is important as it simplifies the study of the interaction between well-definedness and substitutions. In his thesis [82], Mehta presents the following two properties about well-definedness and syntactic replacement:

$$\begin{aligned} \mathcal{D}([x := t]\varphi) &\Rightarrow [x := t]\mathcal{D}(\varphi) \\ [x := t]\mathcal{D}(\varphi) \wedge \mathcal{D}(t) &\Rightarrow \mathcal{D}([x := t]\varphi) \end{aligned}$$

For the rest of this chapter, we may use the following simpler property about well-definedness and idempotent substitutions.

Proposition 3.1. *Let t be a Σ-term. If σ is a substitution then*

$$\mathcal{D}(\sigma(t)) \Leftrightarrow \bigwedge_{x \in \mathcal{V}ar(t)} \mathcal{D}(\sigma(x)) \wedge \sigma(\mathcal{D}(t))$$

The proof of Proposition 3.1 can be found in Appendix A. Idempotent substitutions will be used for the rest of this chapter. Proposition 3.1 will be used to simplify well-definedness formulae.

3.1.3 Conditional Rewriting

In this section, we define the important concept of rewrite rules, and outline their syntactic properties. We will later deal with the semantics of such rules by following a purely syntactic approach (i.e., proofs) within the proof system described in §2.4. The following definitions describe what is meant by a conditional term rewrite rule.

Definition 3.4 (Conditional Identity)**.** A Σ-conditional identity (or simply conditional identity) is a triplet $(l, c, r) \in T_\Sigma \times F_\Sigma \times T_\Sigma$. In this case, l is called the left hand side, r the right hand side, and c the condition of the identity.

The following definition describes the validity of conditional identities.

Definition 3.5 (Valid Conditional Identity)**.** A conditional identity (l, c, r) is valid iff the following sequent is provable

$$c \vdash_{\mathcal{D}} l = r$$

A conditional identity describes an equality between two terms under a certain condition. Note the use of $\vdash_{\mathcal{D}}$. The definition of validity takes into account the presence of ill-defined terms. However, rather counter-intuitively, the following is a valid conditional identity:

$$1 \div 0 = 1 \vdash_{\mathcal{D}} 1 = 0 \tag{3.2}$$

To see why (3.2) is a valid conditional identity according to Definition 3.5, we expand the definition of $\vdash_{\mathcal{D}}$ using the rule $\vdash_{\mathcal{D}} eqv$ described in §2.4.4.2:

$$0 \neq 0, 1 \div 0 = 1 \vdash 1 = 0$$

However, despite the above observation, we will show later that this weak definition of validity is sufficient for our development. A conditional identity can be turned into a rewrite rule if it satisfies the syntactic restrictions presented in the following definition:

Definition 3.6 (Conditional Term Rewrite Rule)**.** A conditional term rewrite rule is a conditional identity (l, c, r) such that:

1. l is not a variable,
2. $\mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$,

3. $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.

In this case, we use the notation $l \xrightarrow{c} r$ instead of (l, c, r). A term rewriting system (TRS) is a set of conditional term rewrite rules.

Definition 3.5 also applies to conditional rewrite rules, because they are essentially conditional identities. The condition that the left hand side of a rewrite rule is not a variable eliminates an obvious non-terminating case [41] (see §3.1.4). The other two conditions ensure that matching can gather sufficient information in order to carry out the rewriting. Matching is the process of matching a term against the left hand side of a rule. It is a special case of unification [65, 98], and given a term t and a left hand side of a rewrite rule l, matching calculates an idempotent substitution σ such that $t = \sigma(l)$. Pattern matching is an important component of theorem proving infrastructure as it provides valuable facilities for equational reasoning. In practice, the matching procedure recursively inspects a formula (considering all possible positions) to establish whether a particular rewrite rule is applicable, and returns the set of positions at which a match was found. For each position, a record of the appropriate idempotent substitution is stored. The precise way in which rewrite rules are applied will be presented and justified in §3.3.

3.1.4 Confluence and Termination

Confluence and termination are important properties of rewrite systems. Confluence describes the property of rewrite systems where terms can be rewritten in different ways to yield the same result. For instance, the rewrite system containing the usual arithmetic is confluent [19]. Termination describes the property of a rewrite system where an infinite rewrite chain may not occur. Central to both concepts of termination and confluence is the notion of term normal form [19].

In a rewriting system, a normal form of a term cannot be rewritten any further. A rewrite system is defined by means of a reduction relation $\rightarrow$ between terms. Given a term t, we write $t \rightarrow t'$, if t can be rewritten to t' by a rule in the rewrite system. We write $t \rightarrow^* t'$ to indicate that there exists a reduction sequence from t to t'. A term t is in normal form if there is no term t' such that $t \rightarrow t'$ [42].

A term t is said to be confluent if for all terms t_1, t_2 such that $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$, there exists a term t' such that $t_1 \rightarrow^* t'$ and $t_2 \rightarrow^* t'$. A rewrite system is said to be confluent if all terms are confluent. A rewrite system is terminating if there is no infinite reduction sequence $t \rightarrow t_1 \rightarrow t_2 \rightarrow ....$ A rewrite system is said to be convergent if it is both confluent and terminating [42, 19].

In this chapter, we do not consider confluence and termination of rewrite rules. Our study is restricted to characterising the interaction between well-definedness and rewriting. This, however, does not hide the fact that termination and confluence are extremely important properties of any rewrite system, including the one considered for Event-B. The large body of research on these two subjects may, in the future, be considered for implementation as part of the Theory plug-in rewriting capabilities (see Chapter 5) .

3.2 Rewriting and Well-Definedness

In the last section, we introduced important concepts in term rewriting. In this section, we present a unifying treatment of well-definedness and rewriting. We show the necessary conditions for a rewrite rule application to preserve well-definedness. The following definition introduces the notion of WD-preserving conditional term rewrite rule.

Definition 3.7 (WD-Preserving Conditional Rewrite Rule)**.** A conditional rewrite rule $l \xrightarrow{c} r$ is said to be WD-preserving if the following sequent is provable:

$$\mathcal{D}(l), c \vdash_{\mathcal{D}} \mathcal{D}(r)$$

Note the use of the well-definedness operator $\mathcal{D}$ (see §2.4.3). In simple terms, a rewrite rule is WD-preserving if the well-definedness of its left hand side is stronger than the well-definedness of its right hand side under the rule's condition. Intuitively, the well-definedness strength relationship corresponds to the directed way in which rewrite rules are applied. In what follows, we describe the significance of WD-preservation in the context of rewriting where undefinedness is an issue. We will show in §3.2.1 how instantiations, i.e., a substitution, interact with well-definedness in the context of Definition 3.7.

Note. In the forthcoming (sub)sections, we carry out a significant proof effort. Traditionally, proofs are presented as trees in a similar fashion to inference rule as per the treatment in §2.4.4.2. Given the complexity of the formulae involved in our proofs, we opt for a clearer approach. We clearly show the sequent to prove. Then, we mention the proof rule which is to be applied. Finally, we show the resulting sequents (if any) from applying the proof rule on the sequent to prove. For example, the following proof step

$$\frac{H \vdash_{\mathcal{D}} P \quad H \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} P \land Q} \land goal_{\mathcal{D}}$$

can be described as follows:

> " In order to show the provability of the sequent
>
> $$H \vdash_{\mathcal{D}} P \land Q$$

we proceed as follows. By applying the rule $\wedge goal_{\mathcal{D}}$, we obtain the following two sequents

$$H \quad \vdash_{\mathcal{D}} \quad P$$
$$H \quad \vdash_{\mathcal{D}} \quad Q\text{"}$$

3.2.1 Well-Definedness and Substitutions

Matching is central to the application of rewrite rules. Given a term t and a left hand side of a rewrite rule l, matching calculates an idempotent substitution σ such that $t = \sigma(l)$. In practice, the matching procedure checks all positions within a formula ϕ for terms which can be matched against the left hand side of a rewrite rule $l \xrightarrow{c} r$. For a particular position $p \in \mathcal{P}os(\phi)$, if a substitution σ is found such that $\phi \mid_p = \sigma(l)$, then rewriting can be executed by replacing $\phi \mid_p$ by the instantiated (using the same substitution) right hand side of the rule, i.e., $\sigma(r)$, within the formula ϕ.

The following theorem formalises the interaction between well-definedness and substitutions in the context of conditional rewrite rules. In the following theorems, we assume the presence of a suitable theory, e.g., Peano axioms for arithmetics.

Theorem 3.1 (The Instantiation Theorem)**.** *Let $l \xrightarrow{c} r$ be a conditional term rewrite rule, and σ be an idempotent substitution.*

1. *If $l \xrightarrow{c} r$ is valid, then the following sequent is provable:*

$$\sigma(c) \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r) \tag{3.3}$$

2. *If $l \xrightarrow{c} r$ is WD-preserving, then the following sequent is provable:*

$$\sigma(c), \mathcal{D}(\sigma(l)) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\sigma(r)) \tag{3.4}$$

The proof of Theorem 3.1 can be found in Appendix A. The Instantiation Theorem concisely describes the interaction of substitutions and well-definedness with respect to conditional rewrite rules. We have shown that instances of the rewrite rule (i.e., with an idempotent substitution) preserve the properties of the rule. This means that if a conditional rewrite rule is valid and well-definedness preserving, then instances created using an idempotent substitution are also valid and well-definedness preserving. The Instantiation Theorem is a building block in our treatment of well-definedness and rewriting since it provides a sound bridge between rewrite rules and their instances in the presence of potentially ill-defined terms. The following theorem states that the application of a valid and well-definedness preserving conditional term rewrite rule preserves equality (A.15) and well-definedness (A.16) of terms.

Theorem 3.2 (Term WD-Preserving Rewriting Theorem)**.** *Let $l \xrightarrow{c} r$ be a conditional term rewrite rule, t be a term, p be a position within t, and σ be an idempotent substitution. If $l \xrightarrow{c} r$ is valid and WD-preserving, then the following two sequents are provable:*

$$\sigma(c) \quad \vdash_{\mathcal{D}} \quad t[\sigma(l)]_p = t[\sigma(r)]_p \tag{3.5}$$

$$\mathcal{D}(t[\sigma(l)]_p), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(t[\sigma(r)]_p) \tag{3.6}$$

The proof of Theorem 3.2 can be found in Appendic A. The Term WD-Preserving Rewriting Theorem states that Definition 3.5 and Definition 3.7 are adequate for a conditional term rewrite rule to preserve equality and well-definedness when applied to a term.

3.2.2 The Main Theorem

In the previous section, we formally described the interaction between idempotent substitutions and well-definedness. The understanding of such interaction is of paramount importance, since in almost all cases, instances of rewrite rules (rather than the actual rewrite rule) occur in practice. Since instances of rewrite rules are obtained by applying an idempotent substitution (i.e., avoiding clashes between rule variables and the actual variables of the instance), it is significant that after applying substitutions, the instances preserve the properties of the conditional rewrite rule. We have also shown that valid and well-definedness preserving rewrite rules ensure valid and well-definiedess preserving rewriting of terms.

We, now, can formulate the main theorem of this chapter. The main theorem asserts that Definition 3.5 and Definition 3.7 are adequate for a conditional term rewrite rule to preserve validity and well-definedness when applied to a formula.

Theorem 3.3 (The Main Theorem)**.** *Let $l \xrightarrow{c} r$ be a conditional term rewrite rule, f be a formula, p be a position within f such that $f|_P$ is a term, and σ be an idempotent substitution. If $l \xrightarrow{c} r$ is valid and WD-preserving, then the following two sequents are provable:*

$$\sigma(c) \quad \vdash_{\mathcal{D}} \quad f[\sigma(l)]_p \Leftrightarrow f[\sigma(r)]_p \ , \tag{3.7}$$

$$\mathcal{D}(f[\sigma(l)]_p), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(f[\sigma(r)]_p) \tag{3.8}$$

with the proviso that all free variables defined as

$$\mathcal{V}ar(\sigma(c)) \ \cap \ \mathcal{V}ar(\sigma(l))$$

do not become bound in $f[\sigma(l)]_p$.

Proof.

1. *Proof of Sequent* 3.7: We proceed by induction on the structure of the formula f. We show a sketch of the proof, and only cover three interesting cases.

   (a) **Base Case:** f is of the shape $s(t_1, ..., t_n)$ such that $s \in P$ and $t_1, ..., t_n$ are terms. In this case, position p can only be of the form iq for some position q and $1 \leq i \leq n$ since the root position is of a formula. Therefore, Sequent 3.7 becomes

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ s(t_1, ..., t_n)[\sigma(l)]_p \Leftrightarrow s(t_1, ..., t_n)[\sigma(r)]_p$$

   where $p = iq$ for some position q and $1 \leq i \leq n$. This can be rewritten to

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ s(t_1, ..., t_i[\sigma(l)]_q, ..., t_n) \Leftrightarrow s(t_1, ..., t_i[\sigma(r)]_q, ..., t_n)$$

   This amounts to proving the following two sequents:

   $$\sigma(c), s(t_1, ..., t_i[\sigma(l)]_q, ..., t_n) \ \vdash_{\mathcal{D}} \ s(t_1, ..., t_i[\sigma(r)]_q, ..., t_n) \tag{3.9}$$
   $$\sigma(c), s(t_1, ..., t_i[\sigma(r)]_q, ..., t_n) \ \vdash_{\mathcal{D}} \ s(t_1, ..., t_i[\sigma(l)]_q, ..., t_n) \tag{3.10}$$

   According to Theorem 3.2, we have the following provable sequent:

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ t_i[\sigma(l)]_q = t_i[\sigma(r)]_q \ .$$

   As such, it is easy to see that Sequent 3.9 and 3.10 are provable.

   (b) **Inductive Case:** f is of the shape $\varphi \wedge \psi$ such that φ and ψ are formulae. In this case, Sequent 3.7 becomes

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ (\varphi \wedge \psi)[\sigma(l)]_p \Leftrightarrow (\varphi \wedge \psi)[\sigma(r)]_p \tag{3.11}$$

   Position p can only be of the form $p = 1q$ or $p = 2q$ for some position q. We distinguish the two cases:

   i. $p = 1q$: In this case, Sequent 3.11 becomes

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ (\varphi[\sigma(l)]_q \wedge \psi) \Leftrightarrow (\varphi[\sigma(r)]_q \wedge \psi) \tag{3.12}$$

   To proceed, we assume the following inductive hypothesis

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ (\varphi[\sigma(l)]_q) \Leftrightarrow (\varphi[\sigma(r)]_q) \tag{3.13}$$

   and we show that Sequent 3.12 is provable. Sequent 3.12 can be reduced to the following two sequents:

   $$\sigma(c), \varphi[\sigma(l)]_q, \psi \quad \vdash_{\mathcal{D}} \quad \varphi[\sigma(r)]_q \wedge \psi$$
   $$\sigma(c), \varphi[\sigma(r)]_q, \psi \quad \vdash_{\mathcal{D}} \quad \varphi[\sigma(l)]_q \wedge \psi$$

which can, respectively, be reduced to the following two sequents:

$$\sigma(c), \varphi[\sigma(l)]_q \quad \vdash_{\mathcal{D}} \quad \varphi[\sigma(r)]_q \tag{3.14}$$

$$\sigma(c), \varphi[\sigma(r)]_q \quad \vdash_{\mathcal{D}} \quad \varphi[\sigma(l)]_q \tag{3.15}$$

It is easy to see that the provability of Sequent 3.14 and Sequent 3.15 follows immediately from the inductive hypothesis i.e., Sequent 3.13.

ii. $p = 2q$: follows by symmetry.

(c) **Inductive Case:** f is of the shape $\forall x \cdot \varphi$ such that φ is a formula. In this case, Sequent 3.7 becomes

$$\sigma(c) \ \vdash_{\mathcal{D}} \ (\forall x \cdot \varphi)[\sigma(l)]_p \Leftrightarrow (\forall x \cdot \varphi)[\sigma(r)]_p \tag{3.16}$$

Position p can only be of the form $p = 1q$ for some position q. Sequent 3.16 simplifies to

$$\sigma(c) \ \vdash_{\mathcal{D}} \ (\forall x \cdot \varphi[\sigma(l)]_q) \Leftrightarrow (\forall x \cdot \varphi[\sigma(r)]_q) \tag{3.17}$$

To proceed, we assume that the following sequent is provable:

$$\sigma(c) \ \vdash_{\mathcal{D}} \ (\varphi[\sigma(l)]_q) \Leftrightarrow (\varphi[\sigma(r)]_q) \tag{3.18}$$

and we show that Sequent 3.17 is provable. Proving Sequent 3.17 amounts to proving the following two sequents:

$$\sigma(c), \forall x \cdot \varphi[\sigma(l)]_q \quad \vdash_{\mathcal{D}} \quad \forall x \cdot \varphi[\sigma(r)]_q \tag{3.19}$$

$$\sigma(c), \forall x \cdot \varphi[\sigma(r)]_q \quad \vdash_{\mathcal{D}} \quad \forall x \cdot \varphi[\sigma(l)]_q \tag{3.20}$$

Proofs for Sequent 3.19 and Sequent 3.20 are similar, and we only show the proof for Sequent 3.19. Firstly, note that the proviso of Theorem 3.3 ensures that universal quantifier does not bind any variables in $\sigma(l)$. This also means x does not occur free elsewhere in Sequent 3.19 and 3.20 since we have the following property that follows from the definition of a conditional rewrite rule:

$$\mathcal{V}ar(\sigma(c)) \ \subseteq \ \mathcal{V}ar(\sigma(l))$$

Now, we can apply the rule $\forall goal_{\mathcal{D}}$ on Sequent 3.19 knowing that its side condition holds. We obtain the following sequent:

$$\sigma(c), \forall x \cdot \varphi[\sigma(l)]_q \quad \vdash_{\mathcal{D}} \quad \varphi[\sigma(r)]_q \tag{3.21}$$

Next, we apply rule $\forall goal_{\mathcal{D}}$ on Sequent 3.21, we get the following two sequents:

$$\sigma(c), \forall x \cdot \varphi[\sigma(l)]_q \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(x) \tag{3.22}$$

$$\sigma(c), \varphi[\sigma(l)]_q \quad \vdash_{\mathcal{D}} \quad \varphi[\sigma(r)]_q \tag{3.23}$$

Sequent 3.22 is provable since variables are well-defined. Sequent 3.23 provability follows from the inductive hypothesis. □

2. *Proof of Sequent* 3.8: is similar to the proof of Sequent 3.7, and is diferred to Appendix A.

□

Theorem 3.3 provides the sufficient conditions under which a conditional rewrite rule preserves validity and well-definedness. Effectively, this section provides a basis for understanding the interaction between rewriting and well-definedness. In the next section, we use the results from this section to show how rewriting can be performed on goal or hypothesis whilst preserving validity and well-definedness. We aim to show how rewriting can be interleaved with deduction in the proof calculus presented in §2.4.4.2.

3.3 Rewriting as a Proof Step

In this section, we show how rewriting can be used in proofs alongside the WD-preserving sequent calculus. In §3.2, we discussed the necessary conditions under which rewriting preserves well-definedness. In what follows, we show by means of proof derivations how rewriting can be integrated into the WD-preserving proof calculus as a proof step. We single out two ways of applying conditional rewrite rules.

3.3.1 Single Rule Application

Let $l \xrightarrow{c} r$ be a valid and WD-preserving conditional term rewrite rule, and σ be an idempotent substitution. The following two subsections describe how rewriting can be applied to hypotheses and the goal.

3.3.1.1 Hypothesis Rewriting

Assume the following sequent whose provability is to be established:

$$H, P[\sigma(l)]_p \vdash_{\mathcal{D}} G \tag{3.24}$$

The hypothesis $P[\sigma(l)]_p$ has an occurrence of an instance of the left hand side of the given rewrite rule. By applying the cut rule on Sequent 3.24 to introduce $\sigma(c)$, we get

the following three sequents:

$$H, P[\sigma(l)]_p \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\sigma(c)) \tag{3.25}$$

$$H, P[\sigma(l)]_p \quad \vdash_{\mathcal{D}} \quad \sigma(c) \tag{3.26}$$

$$H, P[\sigma(l)]_p, \sigma(c) \quad \vdash_{\mathcal{D}} \quad G \tag{3.27}$$

By applying the cut rule on Sequent 3.27, we get the following three sequents:

$$H, P[\sigma(l)]_p, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(P[\sigma(r)]_p) \tag{3.28}$$

$$H, P[\sigma(l)]_p, \sigma(c) \quad \vdash_{\mathcal{D}} \quad P[\sigma(r)]_p \tag{3.29}$$

$$H, P[\sigma(l)]_p, \sigma(c), P[\sigma(r)]_p \quad \vdash_{\mathcal{D}} \quad G \tag{3.30}$$

Sequent 3.28 and Sequent 3.29 are provable thanks to the Main Theorem. Applying hypothesis contraction ($mon_{\mathcal{D}}$) on Sequent 3.30, we get the following sequent

$$H, \sigma(c), P[\sigma(r)]_p \quad \vdash_{\mathcal{D}} \quad G$$

In summary, in order to prove Sequent 3.24, it is sufficient to prove the following three sequents

$$\begin{aligned} H, P[\sigma(l)]_p \quad &\vdash_{\mathcal{D}} \quad \mathcal{D}(\sigma(c)) \\ H, P[\sigma(l)]_p \quad &\vdash_{\mathcal{D}} \quad \sigma(c) \\ H, \sigma(c), P[\sigma(r)]_p \quad &\vdash_{\mathcal{D}} \quad G \end{aligned}$$

Therefore, we have the following proof step:

$$\frac{\begin{cases} H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(c)) \\ H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ \sigma(c) \\ H, \sigma(c), P[\sigma(r)]_p \ \vdash_{\mathcal{D}} \ G \end{cases}}{H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ G}$$

3.3.1.2 Goal Rewriting

Assume the following sequent whose provability is to be established:

$$H \ \vdash_{\mathcal{D}} \ G[\sigma(l)]_p \ , \tag{3.31}$$

The goal $G[\sigma(l)]_p$ has an occurrence of an instance of the left hand side of the given rewrite rule. By applying the cut rule on Sequent 3.31, we get the following three

sequents:

$$H \vdash_{\mathcal{D}} \mathcal{D}(\sigma(c)) \tag{3.32}$$
$$H \vdash_{\mathcal{D}} \sigma(c) \tag{3.33}$$
$$H, \sigma(c) \vdash_{\mathcal{D}} G[\sigma(l)]_p \tag{3.34}$$

By applying the cut rule on Sequent 3.34, we get the following three sequents:

$$H, \sigma(c) \vdash_{\mathcal{D}} \mathcal{D}(G[\sigma(r)]_p) \tag{3.35}$$
$$H, \sigma(c) \vdash_{\mathcal{D}} G[\sigma(r)]_p \tag{3.36}$$
$$H, \sigma(c), G[\sigma(r)]_p \vdash_{\mathcal{D}} G[\sigma(l)]_p \tag{3.37}$$

Sequent 3.35 and Sequent 3.37 are provable thanks to Theorem 3.3. In summary, in order to prove Sequent 3.31, it is sufficient to prove the following three sequents

$$H \vdash_{\mathcal{D}} \mathcal{D}(\sigma(c))$$
$$H \vdash_{\mathcal{D}} \sigma(c)$$
$$H, \sigma(c) \vdash_{\mathcal{D}} G[\sigma(r)]_p$$

Therefore, we have the following proof step:

$$\frac{\begin{cases} H \vdash_{\mathcal{D}} \mathcal{D}(\sigma(c)) \\ H \vdash_{\mathcal{D}} \sigma(c) \\ H, \sigma(c) \vdash_{\mathcal{D}} G[\sigma(r)]_p \end{cases}}{H \vdash_{\mathcal{D}} G[\sigma(l)]_p}$$

3.3.2 Grouped Rule Application

In the previous subsection, we showed how a single rule can be applied to rewrite a hypothesis or a goal of a sequent. In this subsection, we adopt a rather different approach. We present a convenient mechanism to study rewrite rules in the context of proofs. We consider grouping rewrite rules that have the same left hand side. The following definition describes the notion of a grouped conditional term rewrite rule.

Definition 3.8 (Grouped Conditional Term Rewrite Rule)**.** A grouped conditional term rewrite rule is of the form

$$l \rightarrow \quad c_1 : r_1$$
$$\ldots$$
$$c_n : r_n$$

where each of $l \xrightarrow{c_i} r_i$, for $1 \leq i \leq n$, is a conditional term rewrite rule. Moreover, the grouped conditional rewrite rule is said to be valid and well-definedness preserving if each of

$$l \xrightarrow{c_i} r_i$$

for $1 \leq i \leq n$ is a valid and well-definedness preserving conditional rewrite rule. Finally, the grouped conditional rewrite rule is homogeneously simple-conditioned if the following two syntactic properties hold:

1. each of the conditions is a simple formula i.e.,

$$c_i \;\widehat{=}\; f(t_1, ..., t_m)$$

   for some predicate symbol f and terms t_k $(1 \leq k \leq m)$ for all i such that $1 \leq i \leq n$.

2. all conditions include exactly the same set of free variables i.e.,

$$\mathcal{V}ar(c_1) \;=\; ... \;=\; \mathcal{V}ar(c_n)$$

Homogeneously simple-conditioned grouped conditional rewrite rules have important properties with respect to well-definedness. But, first, let us introduce an important property of this class of grouped rewrite rules.

Definition 3.9 (Case-Completeness)**.** A homogeneously simple-conditioned grouped conditional term rewrite rule

$$\begin{aligned} l \rightarrow \quad & c_1 : r_1 \\ & ... \\ & c_n : r_n \end{aligned}$$

is said to be case-complete if the following sequent is provable:

$$\vdash_{\mathcal{D}} \bigvee_{i=1}^{n} c_i$$

Intuitively, a homogeneously simple-conditioned grouped rewrite rule is case-complete if its conditions cover, i.e., a disjunction of, all possible cases under which a rewrite can occur. An important property of homogeneously simple-conditioned grouped rewrite rules is that the interaction between case-completeness and substitutions can be succinctly formulated. The following proposition describes the said interaction.

Proposition 3.2. *Let σ be an idempotent substitution. If the homogeneously simple-conditioned grouped conditional term rewrite rule*

$$\begin{aligned} l \rightarrow \quad & c_1 : r_1 \\ & \ldots \\ & c_n : r_n \end{aligned}$$

is case-complete, then the following sequent is provable:

$$\vdash_{\mathcal{D}} \bigvee_{i=1}^{n} \sigma(c_i) \ . \tag{3.38}$$

Proof. Since the grouped rule is homogeneously simple-conditioned and case complete, the following sequent is provable:

$$\vdash_{\mathcal{D}} \ \forall \vec{x} \cdot \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \Rightarrow \bigvee_{i=1}^{n} c_i \ . \tag{3.39}$$

To show the provability of Sequent 3.39, we proceed as follows. By applying rule $\forall goal_{\mathcal{D}}$, we get the following sequent

$$\vdash_{\mathcal{D}} \ \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \Rightarrow \bigvee_{i=1}^{n} c_i \ .$$

Next, we apply rule $\Rightarrow goal_{\mathcal{D}}$, and we obtain the following sequent

$$\bigwedge_{i=1}^{n} \mathcal{D}(c_i) \ \vdash_{\mathcal{D}} \ \bigvee_{i=1}^{n} c_i \ .$$

Finally, applying hypothesis contraction on the previous sequent, we obtain the sequent

$$\vdash_{\mathcal{D}} \ \bigvee_{i=1}^{n} c_i \ ,$$

which is provable since the grouped rule is case-complete.

Since the rule is homogeneously simple conditioned, the well-definedness of each condition depends only on the well-definedness of the terms occurring in them. By definition, all rule conditions refer to the same set of terms. Hence, the well-definedness of each instantiated condition is the following:

$$\mathcal{D}(\sigma(c_i)) \ \Leftrightarrow \bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x)) \tag{3.40}$$

for each i such that $1 \leq i \leq n$.

To show the provability of Sequent 3.38, we proceed as follows. Firstly, we use rule $goal_{WD}$ to add the well-definedness of the goal. We obtain the following sequent

$$\bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x)) \ \vdash_{\mathcal{D}} \ \bigvee_{i=1}^{n} \sigma(c_i)$$

We apply the cut rule on the previous sequent to introduce the formula

$$\forall \vec{x} \cdot \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \Rightarrow \bigvee_{i=1}^{n} c_i$$

We obtain the following three sequents

$$\bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x)) \vdash_{\mathcal{D}} \mathcal{D}(\forall \vec{x} \cdot \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \Rightarrow \bigvee_{i=1}^{n} (c_i)) \tag{3.41}$$

$$\bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x)) \vdash_{\mathcal{D}} \forall \vec{x} \cdot \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \Rightarrow \bigvee_{i=1}^{n} (c_i) \tag{3.42}$$

$$\bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x)), \forall \vec{x} \cdot \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \Rightarrow \bigvee_{i=1}^{n} (c_i) \vdash_{\mathcal{D}} \bigvee_{i=1}^{n} \sigma(c_i) \tag{3.43}$$

The first two sequents are provable as it follows from our discussion above. Next, by applying rule $\forall hyp_{\mathcal{D}}$ on Sequent 3.43, we obtain the following two sequents

$$\bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x)), \forall \vec{x} \cdot \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \Rightarrow \bigvee_{i=1}^{n} (c_i) \vdash_{\mathcal{D}} \bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x))$$

$$\bigwedge_{x \in \mathcal{V}ar(c_i)} \mathcal{D}(\sigma(x)), \bigwedge_{i=1}^{n} \sigma(\mathcal{D}(c_i)) \Rightarrow \bigvee_{i=1}^{n} \sigma(c_i) \vdash_{\mathcal{D}} \bigvee_{i=1}^{n} \sigma(c_i)$$

The rest of the proof is trivial. □

3.3.2.1 Hypothesis Rewriting

In this section, we show how grouped rewrite rules can be used to rewrite a hypothesis within a sequent. Let

$$\begin{aligned} l \rightarrow \quad & c_1 : r_1 \\ & \ldots \\ & c_n : r_n \end{aligned}$$

be a valid and WD-preserving grouped conditional term rewrite rule. Consider the sequent whose provability is to be established:

$$H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ G \ . \tag{3.44}$$

By applying the cut rule on Sequent 3.44 to introduce the following formula

$$\sigma(c_1) \vee ... \vee \sigma(c_n) \ ,$$

we obtain the following three sequents

$$H, P[\sigma(l)]_p \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\sigma(c_1) \vee ... \vee \sigma(c_n)) \tag{3.45}$$

$$H, P[\sigma(l)]_p \quad \vdash_{\mathcal{D}} \quad \sigma(c_1) \vee ... \vee \sigma(c_n) \tag{3.46}$$

$$H, P[\sigma(l)]_p, \sigma(c_1) \vee ... \vee \sigma(c_n) \quad \vdash_{\mathcal{D}} \quad G \ . \tag{3.47}$$

Next, we apply rule $\vee hyp_{\mathcal{D}}$ on Sequent 3.47, and we obtain the following n sequents

$$\begin{array}{lll} H, P[\sigma(l)]_p, \sigma(c_1) & \vdash_{\mathcal{D}} & G \\ & ... & \\ H, P[\sigma(l)]_p, \sigma(c_n) & \vdash_{\mathcal{D}} & G \ . \end{array}$$

Next, for each sequent i from the above set of sequents, we apply a single rewrite step as discussed in §3.3.1. We get the following n sequents

$$\begin{array}{lll} H, P[\sigma(r_1)]_p, \sigma(c_1) & \vdash_{\mathcal{D}} & G \\ & ... & \\ H, P[\sigma(r_n)]_p, \sigma(c_n) & \vdash_{\mathcal{D}} & G \ . \end{array}$$

In summary, the following grouped rule application can be added as a proof step:

$$\frac{\left\{ \begin{array}{l} H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(c_1) \vee ... \vee \sigma(c_n)) \\ H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ \sigma(c_1) \vee ... \vee \sigma(c_n) \\ H, \sigma(c_1), P[\sigma(r_1)]_p \ \vdash_{\mathcal{D}} \ G \ \ ... \ \ H, \sigma(c_n), P[\sigma(r_n)]_p \ \vdash_{\mathcal{D}} \ G \end{array} \right.}{H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ G} \ \rightarrow hyp_{\mathcal{D}} \tag{3.48}$$

under the proviso that all free variables of $\sigma(c_i)$ (for all i such that $1 \leq i \leq n$) occur free in $P[\sigma(l)]_p$. This proof step allows the hypothesis $P[\sigma(l)]_p$ to be rewritten to several cases according to the rewrite rule.

3.3.2.2 Goal Rewriting

In this section, we show how grouped rewrite rules can be used to rewrite the goal of a sequent. Let

$$\begin{aligned} l \rightarrow \quad & c_1 : r_1 \\ & \ldots \\ & c_n : r_n \end{aligned}$$

be a valid and WD-preserving grouped conditional term rewrite rule. Consider the sequent whose provability is to be established:

$$H \ \vdash_{\mathcal{D}} \ G[\sigma(l)]_p \ . \tag{3.49}$$

By applying the cut rule on Sequent 3.49 to introduce the following formula

$$\sigma(c_1) \vee \ldots \vee \sigma(c_n) \ ,$$

we obtain the following three sequents

$$H \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\sigma(c_1) \vee \ldots \vee \sigma(c_n)) \tag{3.50}$$

$$H \quad \vdash_{\mathcal{D}} \quad \sigma(c_1) \vee \ldots \vee \sigma(c_n) \tag{3.51}$$

$$H, \sigma(c_1) \vee \ldots \vee \sigma(c_n) \quad \vdash_{\mathcal{D}} \quad G[\sigma(l)]_p \ . \tag{3.52}$$

Next, we apply rule $\vee hyp_{\mathcal{D}}$ on Sequent 3.52, and we obtain the following n sequents

$$\begin{aligned} H, \sigma(c_1) \quad & \vdash_{\mathcal{D}} \quad G[\sigma(l)]_p \\ & \ldots \\ H, \sigma(c_n) \quad & \vdash_{\mathcal{D}} \quad G[\sigma(l)]_p \ . \end{aligned}$$

Next, for each sequent i from the above set of sequents, we apply a single rewrite step as discussed in §3.3.1. We get the following n sequents

$$\begin{aligned} H, \sigma(c_1) \quad & \vdash_{\mathcal{D}} \quad G[\sigma(r_1)]_p \\ & \ldots \\ H, \sigma(c_n) \quad & \vdash_{\mathcal{D}} \quad G[\sigma(r_n)]_p \ . \end{aligned}$$

In summary, the following grouped rule application can be added as a proof step:

$$\frac{\left\{ \begin{array}{l} H \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(c_1) \vee \ldots \vee \sigma(c_n)) \\ H \ \vdash_{\mathcal{D}} \ \sigma(c_1) \vee \ldots \vee \sigma(c_n) \\ H, \sigma(c_1) \ \vdash_{\mathcal{D}} \ G[\sigma(r_1)]_p \ \ \ldots \ \ H, \sigma(c_n) \ \vdash_{\mathcal{D}} \ G[\sigma(r_n)]_p \end{array} \right.}{H \ \vdash_{\mathcal{D}} \ G[\sigma(l)]_p} \ \rightarrow goal_{\mathcal{D}} \ . \tag{3.53}$$

under the proviso that all free variables of $\sigma(c_i)$ (for all i such that $1 \leq i \leq n$) occur free in $G[\sigma(l)]_p$. This proof step allows the goal $G[\sigma(l)]_p$ to be rewritten to several cases according to the rewrite rule.

In the following subsections (§3.3.2.3, §3.3.2.4 and §3.3.2.5), we enumerate special cases that occur often in proofs.

3.3.2.3 Unconditional Term Rewrite Rules

A conditional term rewrite rule $l \xrightarrow{c} r$ is called *unconditional* if $c \mathrel{\widehat{=}} \top$. In this case, proof steps (3.48) and (3.53) can be simplified as follows:

$$\frac{H, P[\sigma(r)]_p \ \vdash_{\mathcal{D}} \ G}{H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ G} \ \rightarrow uhyp_{\mathcal{D}} \tag{3.54}$$

$$\frac{H \ \vdash_{\mathcal{D}} \ G[\sigma(r)]_p}{H \ \vdash_{\mathcal{D}} \ G[\sigma(l)]_p} \ \rightarrow ugoal_{\mathcal{D}} \ . \tag{3.55}$$

3.3.2.4 Case-complete Grouped Term Rewrite Rules

If the grouped term rewrite rule

$$\begin{aligned} l \rightarrow \quad & c_1 : r_1 \\ & \ldots \\ & c_n : r_n \end{aligned}$$

is homogeneously simple-conditioned and case-complete, then proof steps (3.48) and (3.53) can be simplified as follows:

$$\frac{\left\{ \begin{array}{l} H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(c_1) \vee \ldots \vee \sigma(c_n)) \\ H, \sigma(c_1), P[\sigma(r_1)]_p \ \vdash_{\mathcal{D}} \ G \ \ \ldots \ \ H, \sigma(c_n), P[\sigma(r_n)]_p \ \vdash_{\mathcal{D}} \ G \end{array} \right.}{H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ G} \ \rightarrow chyp_{\mathcal{D}} \tag{3.56}$$

$$\frac{\left\{ \begin{array}{l} H \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(c_1) \vee \ldots \vee \sigma(c_n)) \\ H, \sigma(c_1) \ \vdash_{\mathcal{D}} \ G[\sigma(r_1)]_p \ \ \ldots \ \ H, \sigma(c_n) \ \vdash_{\mathcal{D}} \ G[\sigma(r_n)]_p \end{array} \right.}{H \ \vdash_{\mathcal{D}} \ G[\sigma(l)]_p} \ \rightarrow cgoal_{\mathcal{D}} \ . \tag{3.57}$$

3.3.2.5 Strict Term Occurrence

In this section, we present a special case where rewriting can be further simplified. An operator is strict if its well-definedness requires the well-definedness of all its arguments.

Definition 3.10 (Strict Term Occurrence)**.** Let t be a term, f be a formula, p be a position within f. We say that t has a strict occurrence p in f if f is either of the form

1. $q(t_1, ..., t_n)[t]_p$ where $q \in P$ and t_1,..., t_n are terms, or;
2. $(t_1 = t_2)[t]_p$ where t_1 and t_2 are terms.

If t has a strict occurrence in f, then it also has a strict occurrence in $\neg f$.

We have the following interesting property:

Proposition 3.3. *If the term t has a strict occurrence in formula f, then the following holds*

$$\vdash_{\mathcal{D}} \ \mathcal{D}(f) \Rightarrow \mathcal{D}(t) \ .$$

If we further constrain grouped conditional term rewrite rules such that we have

$$\vdash_{\mathcal{D}} \ \mathcal{D}(l) \Rightarrow \bigwedge_{i=1}^{n} \mathcal{D}(c_i) \ ,$$

Proposition 3.3 can be used to simplify proofs. Let $P[\sigma(l)]_p$ be a formula such that $\sigma(l)$ has a strict occurrence. Since the grouped term rewrite rule is valid and WD-preserving, and using the previous proposition, we have the following

$$\begin{aligned} \mathcal{D}(P[\sigma(l)]_p) &\Rightarrow \mathcal{D}(\sigma(l)) \\ &\Rightarrow \bigwedge_{i=1}^{n} \mathcal{D}(\sigma(c_i)) \end{aligned}$$

under the proviso that all free variables of $\sigma(c_i)$ (for all i such that $1 \leq i \leq n$) occur free in $P[\sigma(l)]_p$. In this particular case, the sequents

$$\begin{gathered} H, P[\sigma(l)]_p \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(c_1) \vee ... \vee \sigma(c_n)) \ , \\ H, P \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(c_1) \vee ... \vee \sigma(c_n)) \end{gathered}$$

in (3.48) and (3.53) respectively, are guaranteed to be discharged. As such, they could be removed from the list of sub-goals that the modeller sees.

3.4 Related Work

The interleaving between deduction and rewriting steps has gathered much interest given its importance to automated reasoning. In this work, we identify the necessary conditions under which rewriting can interleave with deduction in the proof calculus defined in [81]. In other works, this interleaving is studied from different perspectives.

Theorem proving modulo [45] is an approach that removes computational steps from proofs by reasoning modulo a congruence on propositions. The advantage of this technique is that it separates computation steps (i.e., rewriting) from deduction steps in a clean way. In [45], a proof-theoretic account of the combination between computations and deductions is presented in the shape of a sequent calculus modulo. The congruence on propositions, on the other hand, is defined by rewrite rules and equational axioms. From the author's experience of the Rodin tool, rewriting represents a considerable contributor to proof effort (on average two thirds of the proof steps are rewrites). This makes rewriting an important component of the Event-B tootlset. In Chapter 4, we show how the results of this chapter enabled the provision of an extensible mechanism for defining, validating and using rewrite rules in Rodin.

The combination of rewriting and deduction makes properties of rewrite systems of practical interest. Termination and confluence properties of term rewriting systems are important, and have been studied extensively [19, 42]. When rewriting is interleaved with deduction, it is critical that computation steps terminate. Term orderings, in which any term that is syntactically simpler that another is smaller than the other, provides a practical technique to assess the termination of rewrite systems.

In our work, we aim to unify the notions of well-definedness and rewrite systems. Our objective is to characterise the interaction between deduction and rewriting when well-definedness is taken into consideration. This is achieved by identifying the necessary conditions under which computations can interleave with the deduction steps (i.e., proof rules) in [81].

Schmalz devised a foundation of 'directed rewriting' for logics supporting partial functions [103]. His treatment starts with assuming a three-valued semantics logic, and uses a shallow Isabelle/HOL embedding to reason about the various components of the logic. He explains how conditional directed rewrite rules can be applied within proofs and justifies the soundness of their application. Directed rewriting is unsafe in general, i.e., it may transform a provable statement into an unprovable one and thus lead a proof attempt into a dead end. However, an approach to avoid this unsafety is also presented in [103]. Furthermore, the author claims that directed rewriting significantly reduces the number of well-definedness checks required during proofs. Schmalz approach reached similar conclusions regarding the sufficient conditions for maintaining soundness when rewriting is performed. In fact, the notion of directed rewriting is similar to the concept

of well-definedness preserving rewriting where rewrite rules preserve well-definedness in one direction, i.e., left to right. A major difference between our approach and [103] is the fact that our approach is entirely syntactic as we do not depart from the syntactic manipulation of proofs.

3.5 Summary

In this chapter, we defined the criteria for the validity and well-definedness preservation of term rewrite rules when rewriting interleaves with the rules of the proof system developed in [81]. We started our discussion by presenting term rewriting systems. A cornerstone in our treatment is the notion of well-definedness preserving conditional rewrite rules. We have shown that valid and well-definedness preserving rewrite rules can soundly be used within the the well-definedness preserving proof system. We precisely described and justified how individual and grouped rewrite rules can be used as well-definedness preserving proof steps. This chapter is the main theoretical contribution of this thesis, and includes a major proof effort to establish the necessary theorems. The work in this chapter complements Mehta's work in [81] to provide a proof system that includes both deduction and computation steps. In the next chapter, we show how the theoretical results presented in this chapter influenced our approach to prover extensibility in Event-B.

Chapter 4

A Practical Approach to Event-B Prover and Language Extensibility

In the previous chapter, we presented the treatment of rewriting within a proof system that admits potentially ill-defined terms. In this chapter, we present a contribution of a more practical inclination. We discuss our approach in dealing with issues related to prover (§2.3.4) and language (§2.3.3) extensibility in Event-B. This chapter is motivated by the discussion in §1.1. Together with Chapter 5, it achieves objectives (1), (2b) and (3) listed in §1.2, and sheds light on the practical contributions of this thesis. Chapter 3 provides the theoretical backbone to the work on rewriting in §4.4.

As mentioned in §1.2, an important requirement of such approach is the practicality of use. More importantly, a mechanism must be in place to avoid compromising the soundness of the formalism. Dealing with prover extensibility becomes a more pressing issue when support for extending the Event-B mathematical language is in place. Specifying new operators and datatypes requires the provision of a mechanism to reason about such extensions when used in Event-B models. As such, we argue that support for language extensibility goes hand in hand with support for prover extensibility.

This chapter is structured in the following way. We start our discussion by recalling the limitations of the existing infrastructure that triggered the need for our work. Next, we present the *theory construct* which is the vehicle we use to specify and reason about extensions. Then, we outline the three possible mechanisms by which the prover can be augmented. We present rewriting as a proof step in Rodin. We also discuss the addition of polymorphic theorems and inference rules. Next, we present how new polymorphic operators can be specified. Operator properties such as associativity and commutativity are discussed. Next, datatype extensions are introduced in terms of the appropriate syntactic restrictions placed on them. Then, we show how primitive recursive operators

can be defined on datatypes. We conclude this chapter by discussing the related work that influenced our approach in dealing with prover and language extensibility. The work presented in this chapter is a continuation of the effort described in [78].

4.1 The Existing Infrastructure

The Rodin platform provides a proof infrastructure that is highly optimised for proof engineering and reuse. Mehta provides a succinct description of the said infrastructure in his thesis [82]. However, prior to our work, the architecture had the limitations discussed in §2.3.4.

External provers can be plugged into the proof infrastructure. Examples of such additions include ML and PP [7]. Other recent efforts include an SMT solver [5] and an Isabelle/HOL translator/prover [101]. ML and PP do not provide sufficient information about how the proof of a sequent has been achieved. ML and PP run as external processes to Rodin, and only return a success or a failure status without providing a proof trace to Rodin. Besides, information such as the set of needed hypotheses is important for proof *reuse* and *replay* [82]. Those properties of proofs are crucial to an efficient running of a reactive modelling environment.

4.1.1 The Existing Constructs

Modelling in Event-B is carried out by means of contexts and machines as discussed in §2.2.2. Contexts are used to specify the static properties of the system to model. Contexts have the general layout depicted in Figure 4.1. Modellers can specify theorems as part of contexts to ensure that the axioms capture their intentions. Appropriate proof obligations are generated to ensure theorems are well-defined and valid. Machines, on

context *name*
carrier sets $S_1, ..., S_n$
{ $\langle Constant \rangle$
| $\langle Axiom \rangle$}

Figure 4.1: Context Structure

the other hand, are used to specify the dynamic properties of the system. Machines have the general layout depicted in Figure 4.2. We argue that contexts and machines are not suitable for defining prover and language extensions for the following two reasons:

1. Contexts and machines are modelling vehicles. They are intended for specifying and reasoning about models of complex systems. As such, they should not be overloaded to specify and *meta-reason* about mathematical and prover extensions.

machine *name*
{ ⟨*Refines Clause*⟩
| ⟨*Sees Clause*⟩
| ⟨*Variable*⟩
| ⟨*Invariant*⟩
| ⟨*Variant*⟩
| ⟨*Event*⟩}

Figure 4.2: Machine Structure

2. Contexts have been used to define useful structures axiomatically, e.g., [99], and to facilitate proof by supporting theorems. However, their intended use was to parametrise machines [60]. As such, an objective of our work is to simplify the use of contexts by providing a third construct independent from contexts and machines in order to separate concerns, handle and meta-reason about extensions. The new construct is called a *theory*. Using our approach, contexts act as a parametrisation mechanism for machines, and theories act as a placeholder for extensions.

4.2 The Theory Construct

Theories [78] are Event-B constructs which are similar in their morphology to contexts and machines. The name of the construct is based on a similar concept in the Isabelle theorem prover [94]. Theories in Event-B, however, differ in purpose from Isabelle theories. Isabelle theories can be used to specify mathematical theories as well as entire logics such as higher-order logic. The notions of inner and outer syntax [94] refer to the object logic and the meta-logic, respectively. A theory in Event-B, on the contrary, is only used for meta-reasoning about the Event-B mathematical language. A theory acts as a place-holder for mathematical and prover extensions. The following listing describes the overall structure of Event-B theories.

theory *name*
imports $t_1, ..., t_n$
type parameters $T_1, ..., T_n$
{ ⟨*Datatype Definition*⟩
| ⟨*Operator Definition*⟩
| ⟨*Polymorphic Theorem*⟩
| ⟨*Metavariables*⟩
| ⟨*Rewrite Rule*⟩
| ⟨*Inference Rule*⟩ }

Figure 4.3: The Theory Construct

An Event-B theory has a name which identifies it within the workspace. Hierarchies of theories can be created by means of the import directive. 'Theory ***A*** imports theory ***B***'

indicates that all definitions and rules of theory ***B*** can be used in theory ***A***. A theory can have an arbitrary number of type parameters which are sets that are assumed to be non-empty and pairwise distinct in which case the theory is said to be *polymorphic* on its type parameters. A theory may also contain an arbitrary number of definitions and rules. In the subsequent sections of this chapter, we describe proof rules and polymorphic theorems and show how they can be specified and validated through the theory construct. Figure 4.4 summarises the new anatomy of Event-B models (as opposed to the old anatomy in Figure 2.1) as a result of the introduction of the new theory component.

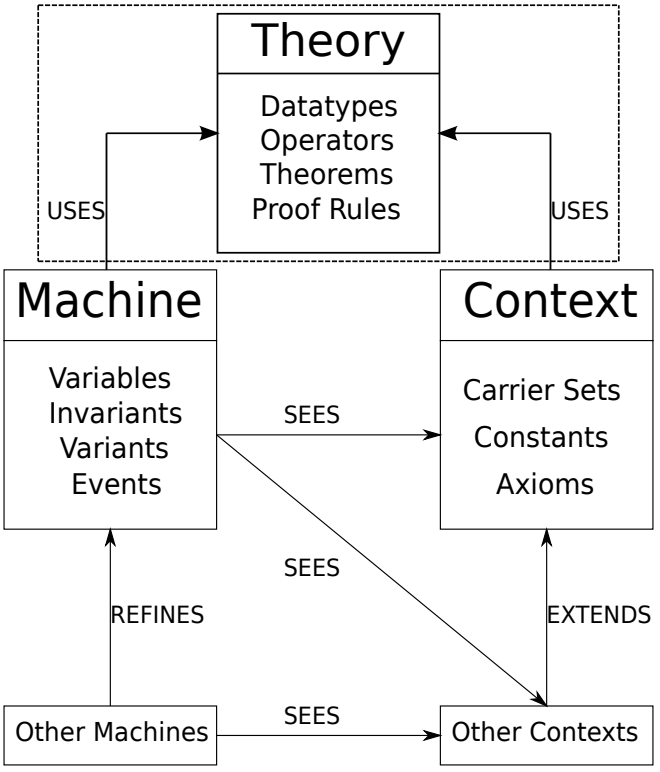


Figure 4.4: Extended Anatomy of Event-B Models

4.2.1 Soundness Preservation

In the process of defining new extensions (e.g., new operator or a new rewrite rule), it is possible to introduce unsoundness to the prover. As such, it is imperative that the ease of use of the theory component is complemented by an effective measure to discover and eliminate any soundness-threatening extensions. Furthermore, we argue that such measure should not hinder the usability of any provided tool support.

The use of proof obligations is widespread in many formal techniques not least in Event-B. In the case of Event-B modelling, proof obligations provide simple semantics by which it is possible to understand the system being modelled [61]. We argue that using proof obligations to verify any user-defined extensions will ensure that potentially unsound

extensions are brought to the attention of the user. Moreover, since modellers are *familiar* with the use of proof obligations in contexts and machines, this approach achieves a good balance between effectiveness and usability. As such, the overhead of proofs in theories can be similar to that of proofs in models. However, the polymorphic nature of theories enables the reusability of proofs, e.g., defining and proving a polymorphic theorem once in a theory and then using it multiple times in different models without the need for reproving it. Throughout the rest of this chapter and subsequent chapters, whenever a new extension is introduced, any required proof obligations are singled out and their adequacy is justified.

4.2.2 Theory Deployment

We distinguish between two separate but intrinsically linked activities in the context of Event-B theories. *Theory development* refers to the activity of defining and validating theories. At this stage, extensions are defined and proof obligations are automatically generated for each extension as required. This activity may follow an iterative pattern since inspecting failed automatic proof attempts may reveal important information about the soundness or otherwise of extensions. Performing interactive proofs provides feedback and guides the modeller to change definitions if appropriate. Therefore, theory development greatly benefits from the reactive nature of the Rodin platform[80, 12].

Theory deployment refers to the activity of making developed theories available for use in modelling. A theory can be used by many models at the same time, thus promoting reusability. Theory deployment ensures that proof obligations are at least inspected by the user, and once deployed, any mathematical extensions and proof rules can be used to specify Event-B contexts and machines. As an example, consider a theory of boolean operators. The user may specify the usual operators (e.g., logical AND), define some inference and rewrite rules, and attempt to discharge any generated proof obligations. Once the user discharged all generated proof obligations, the theory can be deployed and used within a model that specifies an electric circuit. The use of theory-defined proof rules and polymorphic theorems enables the user to reason at the level of mathematical extensions without detour through the existing Event-B mathematical language by means of purpose-built proof tactics.

4.3 Event-B Mathematical Language

In the Event-B mathematical language [83] (Event-B inner syntax), terms (expressions) and formulae (predicates) are separate syntactic categories. Terms are defined using constants (e.g., 1), variables and operators (e.g., $\cup$). Term operators can have terms as arguments. They can also have formulae as arguments e.g., $(\lambda x \cdot P(x) \mid E(x))$ where $P(x)$ is a formula and $E(x)$ is a term.

Formulae, on the other hand, are built from basic formulae e.g., $x \in S$, logical connectives and quantifiers. Basic formulae take terms as arguments e.g., $x \in S$ has x and S as arguments.

Terms have a type which can be one of the following:

1. a basic set such as $\mathbb{Z}$ or a carrier set supplied by the modeller in contexts;
2. a power set of another type;
3. a cartesian product of two types.

Term operators have typing rules of the form:

$$\frac{\mathbf{type}(x_1) = \alpha_1 \ ... \ \mathbf{type}(x_n) = \alpha_n}{\mathbf{type}(op(x_1, ..., x_n)) = \alpha}$$

Arguments of a basic formula must satisfy its typing rule e.g., the typing rule for the basic formula $finite(R)$ is:

$$\mathbf{type}(R) = \mathbb{P}(\alpha)$$

Alongside typing rules, term operators have well-definedness formulae. $\mathcal{D}(E)$ is used to denote the well-definedness formula of term E. Proof obligations are generated (if necessary) to establish the well-definedness of terms appearing in models. To illustrate, we consider the term $card(E)$ for which we have:

$$\mathcal{D}(card(E)) \quad \Leftrightarrow \quad \mathcal{D}(E) \ \wedge \ finite(E)$$

Note. For the rest of this thesis, we use the term *'mathematical language'* to refer to Event-B inner syntax that is wired. The term *'existing mathematical language'* refers to the mathematical language augmented with any previously defined operator extensions.

4.4 Rewriting

The use of equations is central to mathematics. Rewriting provides a powerful mechanism for 'dealing computationally with equations' [43]. In this section, we show how rewrite rules are defined in the theory construct. We also present the different proof obligations that ensure soundness of defined rules. The theoretical results in Chapter 3 provide the justification for the proof obligations related to rewrite rule definitions. At the end of this section, we provide some examples of rewrite rules.

4.4.1 Defining Rewrite Rules

In the Event-B mathematical language [11, 83], formulae and terms are distinguished as two separate syntactic categories[1]. Furthermore, each term must have a type. We say that two Event-B legal syntactic tokens are of the same syntactic class if they are both terms or both formulae. The following definition describes the syntactic properties of Event-B rewrite rules.

Note. Note the introduction of the typing constraint. Also note that rewrite rules left hand sides may contain terms or basic formulae as long as the formulae is built up from *strict* operators.[2]

Definition 4.1 (Event-B Rewrite Rule)**.** An Event-B rewrite rule is of the form

$$\begin{aligned} lhs \rightarrow \quad & C_1 : rhs_1 \\ & \ldots \\ & C_n : rhs_n \end{aligned}$$

where:

1. $n \geq 1$,
2. lhs is not a variable,
3. lhs and rhs_i (for all i such that $1 \leq i \leq n$) are of the same syntactic class,
4. C_i (for all i such that $1 \leq i \leq n$) are formulae,
5. C_i and rhs_i (for all i such that $1 \leq i \leq n$) only contain free variables from lhs,
6. lhs and rhs_i (for all i such that $1 \leq i \leq n$) have the same type if lhs is a term.

In the special case where $n = 1$ and C_1 is syntactically equal to $\top$, the rewrite rule is called *unconditional*. An Event-B rewrite rule is said to be *conditional* if it is not unconditional. Note that Event-B does not have the notion of sorts, and types in Event-B are assumed to be maximal. For instance, $\mathbb{Z}$ is the type for integers, and the set of natural numbers, $\mathbb{N}$, is not a type since it is a subset of $\mathbb{Z}$, and hence not maximal. Given the absence of sorts in Event-B, sort-related properties are not relevant to our discussion of rewriting.

A definition of a rewrite rule is completed by specifying whether the rule should be applied automatically or interactively. In §4.4.4, we describe certain cases where an

[1] In [83], terms are referred to as expressions, and formulae are referred to as predicates.

[2] Strict operators are operators whose well-definedness requires the well-definedness of all its arguments. Hence, if an argument of a strict operator is not well-defined, the expression/predicate rooted at that operator is also not well-defined. For example, equality in Event-B is a strict operator, whereas the conjunction operator ($\wedge$) is not.

automatic application should not be allowed. Figure 4.5 depicts the general layout for a rewrite rule as part of the theory component, where $P(x_1, ..., x_n)$ provides typ-

rewrite *name*
[automatic] [interactive] [case complete]
vars $x_1, ..., x_n$
condition $P(x_1, ..., x_n)$
lhs $lhs(x_1, ..., x_n)$
rhs

$C_1(x_1, ..., x_n)$	$rhs_1(x_1, ..., x_n)$
...	...
$C_m(x_1, ..., x_n)$	$rhs_m(x_1, ..., x_n)$

Figure 4.5: Rewrite Rule Definition

ing information for each of the variables x_i occurring in the left hand side of the rule $lhs(x_1, ..., x_n)$, and m is the number of rule right hand sides.

4.4.2 Validating Rewrite Rules

In the previous section, we defined the syntax of Event-B rewrite rules. In this subsection, we describe the different proof obligations that ensure soundness of defined rewrite rules. We recall from §3.3.2 the notions of *case-complete* and *homogeneously simple-conditioned* rewrite rules.

Case-completeness is only really useful in the case of homogeneously simple-conditioned rules (as reflected in Proposition 3.2 in Chapter 3). Note that case-completeness is not a syntactic property, but one that requires mathematical proof in all cases except when the rule is unconditional. The following definition defines soundness in the context of Event-B rewrite rules.

Definition 4.2 (Sound Event-B Rewrite Rule)**.** An Event-B rewrite rule

$$\begin{aligned} lhs \rightarrow \quad & C_1 : rhs_1 \\ & ... \\ & C_n : rhs_n \end{aligned}$$

is said to be sound if the following sequents are provable:

1. $H, \mathcal{D}(lhs), C_i \vdash_{\mathcal{D}} \mathcal{D}(rhs_i)$ for all i such that $1 \leq i \leq n$,
2. (a) $H, C_i \vdash_{\mathcal{D}} lhs = rhs_i$ for all i such that $1 \leq i \leq n$ if lhs is a term, or;
   (b) $H, C_i \vdash_{\mathcal{D}} lhs \Leftrightarrow rhs_i$ for all i such that $1 \leq i \leq n$ if lhs is a formula,

3. (a) $H \vdash_{\mathcal{D}} \bigvee_{i=1}^{n} C_i$ if the rule is case-complete and homogeneously simple conditioned,

   (b) $H, \mathcal{D}(lhs) \vdash_{\mathcal{D}} \mathcal{D}(C_i)$ for all i such that $1 \leq i \leq n$ if the rule is case-complete and homogeneously simple-conditioned.

where H is a formula providing typing information for all free variables occurring in lhs.[3] If the rewrite rule is not case-complete, the proof obligations 3a and 3b in the previous definition are not required.

4.4.3 Applying Rewrite Rules

Chapter 3 explored two ways for applying rewrite rules. Single rule application (§3.3.1) describes how a single conditional rewrite rule can be applied to goals or hypotheses of sequents. Grouped rule application (§3.3.2) describes how rules sharing the same left hand side can be soundly applied to sequents. Soundness is not the only issue regarding rule application. Termination of automatic rewriting is another potentially problematic aspect of rule application. Whether to apply a rule automatically or interactively is a serious question that requires pondering. In our approach, the decision whether to apply a rule automatically or interactively rests with the specifier of the rule. Admittedly, this could be a dangerous practise. However, we argue that the following guidelines may help with deciding whether a rule should be considered for automatic application:

- The syntactic restriction on rules which states that a left hand side may not be a variable eliminates a certain non-termination case. For instance, the following rule is not allowed:
$$x \rightarrow \top : x + 0$$
- Rewrite rules that simplify formula and reduce their size should be considered for automatic application.
- Rewrite rules that inflate formulae (e.g., the multiplication distribution over addition rewrite rule) should be considered for interactive application. Such rules are more likely to lead to more complicated proofs.
- Care should be taken when defining rules for commutative operators such as $+$. For example, the following rule
$$x + y \rightarrow \top \ : \ y + x$$
should not be applied automatically as it leads to a non-terminating rewriting.

[3]The typing information is provided by the user as part of rule specification in theories.

- Rewrite rules that are considered for automatic application may also be considered for interactive application. This is particularly useful if the user of Rodin decided to turn off automatic provers.
- Definitional rules (i.e., rules that define an operator e.g., the union operator) should be considered for interactive application. Instead of expanding operator definitions, reasoning about operators should be carried out using specially-written proof rules.

4.4.4 Examples of Rewrite Rules

Example 1. Assuming two variables x and y of the same type $\mathbb{Z}$, then the following

$$\begin{array}{rll} (x-1)(y-1) \rightarrow & x = 1 & : \ 0 \\ & y = 1 & : \ 0 \end{array}$$

is a rewrite rule which is sound but not case-complete. Since the rule has more than one right hand side, it is conditional.

Example 2. Assuming two variables a and b of the same type $\mathbb{Z}$, then the following

$$\begin{array}{rll} card(a..b) \rightarrow & a > b & : \ 0 \\ & a \leq b & : \ b - a + 1 \end{array}$$

is a homogeneously simple-conditioned case-complete sound rewrite rule. Note that $a..b$ denotes the integer range defined as follows

$$a..b \ \widehat{=} \ \{x \cdot a \leq x \leq b\}$$

Example 3. Assuming two type parameters A and B, consider the following conditional rewrite rule:

$$\begin{array}{rll} (f \lhd\!\!\!\!- \{y \mapsto z\})(x) \rightarrow & x = y & : \ z \\ & x \neq y & : \ f(x) \end{array}$$

where

$$\begin{array}{rcl} f & \in & A \leftrightarrow B \\ x & \in & A \\ y & \in & A \\ z & \in & B \end{array}$$

Note that the symbol $\lhd\!\!+$ signifies relational override (as defined in [11], p.328). The aforementioned rule is a homogeneously simple-conditioned case-complete conditional rewrite rule. However, the rule is not sound as it does not preserve well-definedness. Functional application in Event-B requires the function to be functional on the entirety of its domain. In the above rewrite rule, $f \lhd\!\!+ \{y \mapsto z\}$ may be functional, but f on its own may not. Therefore, well-definedness is not preserved from the left hand side $(f \lhd\!\!+ \{y \mapsto z\})(x)$ to the second right hand side $f(x)$. To see that this is the case, consider the following instantiations for f, x, y and z:

$$\begin{aligned} f &= \{1 \mapsto 2, 1 \mapsto 3, 2 \mapsto 4\} \\ x &= 1 \\ y &= 1 \\ z &= 3 \end{aligned}$$

It can be seen that $(f \lhd\!\!+ \{1 \mapsto 3\})(1)$ is well-defined and is equals to 3, whereas $f(1)$ is not well-defined as f is not functional at 1.

Summary. In this section, we have shown how new rewrite rules are specified (§4.4.1) in the theory construct. We also discussed the sufficient proof obligations to ensure that soundness of the formalism is not compromised by the addition of new rewrite rules (§4.4.2). We outlined a few guidelines that are helpful in determining whether a rule is suitable for automatic application (§4.4.3) given that termination of rewriting is a major concern.

4.5 Polymorphic Theorems

In an Event-B context, a modeller can specify some static properties of the system in question by means of carrier sets, constants and axioms. In order to ensure that these static properties capture the intended understanding of the system, theorems can be defined in contexts. Similarly, when specifying the dynamic aspects of a system in a machine, certain invariants can be tagged as theorems to verify that the previously added invariants sufficiently restrict the system. The theorems defined in such way are model-specific and more importantly are not *polymorphic*. We propose the addition of polymorphic theorems to the theory component to achieve the following two objectives:

1. package important and reusable properties of pre-defined operators in a succinct and a verifiably sound way, and
2. verify that definitions of any newly introduced operator definitions (see §4.7) capture the intended understanding of the modeller.

Moreover, we describe a mechanism by which these polymorphic theorems can be incorporated in proofs which arise from models. We conclude this section by providing concrete examples.

4.5.1 Defining Polymorphic Theorems

Polymorphic theorems are special Event-B formulae where all variables except type variables (i.e., type parameters) are bound. Intuitively, we envisage polymorphic theorems to be used in proofs in the following way:

1. the modeller chooses the theorem to incorporate into his proof from a collection of theorems,
2. the modeller provides type instantiations appropriate to the current sequent to prove, and the theorem gets instantiated with said type instantiations and added to the set of hypotheses of the sequent.

The following definition describes the syntactic properties satisfied by polymorphic theorems.

Definition 4.3 (Event-B Polymorphic Theorem)**.** Let $\alpha_1, ...,$ and α_n be type parameters. A formula $P(\alpha_1, ..., \alpha_n)$ is an Event-B polymorphic theorem if

$$\mathcal{V}ar(P(\alpha_1, ..., \alpha_n)) \ = \ \{\alpha_1, ..., \alpha_n\}$$

In this case, we say that the theorem $P(\alpha_1, ..., \alpha_n)$ is polymorphic on each of the type parameters $\alpha_1, ...,$ and α_n.

In other words, an Event-B formula is a polymorphic theorem if its free variables are all type parameters.

4.5.2 Validating Polymorphic Theorems

Definition 4.3 describes the syntactic properties of polymorphic theorems. The following definition presents the notion of soundness in the context of polymorphic theorems.

Definition 4.4 (Sound Event-B Polymorphic Theorem)**.** An Event-B polymorphic theorem $P(\alpha_1, ..., \alpha_n)$ is said to be sound if the following sequents are provable:

1. $\vdash_{\mathcal{D}} \ \mathcal{D}(P(\alpha_1, ..., \alpha_n))$
2. $\vdash_{\mathcal{D}} \ P(\alpha_1, ..., \alpha_n)$

Definition 4.4 ensures that polymorphic theorems are well-defined and valid. Note the similarity between the sequents in Definition 4.4 and the proof obligations related to theorems in Event-B contexts, in §2.2.2.1 and [11]. The next sub-section provides a justification for the previous definition.

4.5.3 Using Polymorphic Theorems

In §2.4.4.2, we described the inference rules used in Event-B proofs. The cut rule in particular,

$$\frac{H \vdash_{\mathcal{D}} \mathcal{D}(P) \quad H \vdash_{\mathcal{D}} P \quad H, P \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} Q}\ cut_{\mathcal{D}}$$

can be extremely useful when conducting proofs as it imitates the general approach taken when doing proofs in mathematics (i.e., using intermediate lemmas to guide proofs). In what follows, we show how the cut rule can provide a sound platform for using polymorphic theorems in Event-B proofs. Firstly, we introduce type substitutions which are the cornerstone for using polymorphic theorems.

Definition 4.5 (Type Substitution)**.** A type substitution σ_t consists of a sequence type variables (parameters) mapped to a sequence (of the same length) of types. The domain of σ_t is the set of type variables mapped by the type substitution.

A formula P' is said to be an *instance* of the polymorphic theorem $P(\alpha_1, ..., \alpha_n)$ if there exists a type substitution σ_t such that:

$$P' \ \widehat{=}\ \sigma_t(P(\alpha_1, ..., \alpha_n)) \tag{4.1}$$

where σ_t provides a substitution for all type parameters occurring in $P(\alpha_1, ..., \alpha_n)$. An instance of a polymorphic theorem can be added as a hypothesis in a sequent as follows:

$$\frac{\boxed{H \vdash_{\mathcal{D}} \mathcal{D}(\sigma_t(P(\alpha_1, ..., \alpha_n)))} \quad \boxed{H \vdash_{\mathcal{D}} \sigma_t(P(\alpha_1, ..., \alpha_n))} \quad H, \sigma_t(P(\alpha_1, ..., \alpha_n)) \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} Q}\ cut_{\mathcal{D}}$$

If the polymorphic theorem $P(\alpha_1, ..., \alpha_n)$ is sound as per Definition 4.4, then the boxed sequents can be removed and the polymorphic theorem can be used in proofs as follows:

$$\frac{H, \sigma_t(P(\alpha_1, ..., \alpha_n)) \vdash_{\mathcal{D}} Q}{H \vdash_{\mathcal{D}} Q}\ thm_{\mathcal{D}}$$

4.5.4 Examples

Example 1. The following formula is a sound Event-B polymorphic theorem:

$$\forall x : \mathbb{Z}, y : \mathbb{Z} \cdot x * y = 0 \Rightarrow (x = 0 \vee y = 0)$$

as it can be shown that the following two sequents are provable; hence satisfying the conditions stipulated in Definition 4.4.

$$\vdash_{\mathcal{D}} \quad \mathcal{D}(\forall x : \mathbb{Z}, y : \mathbb{Z} \cdot x * y = 0 \Rightarrow (x = 0 \lor y = 0))$$
$$\vdash_{\mathcal{D}} \quad \forall x : \mathbb{Z}, y : \mathbb{Z} \cdot x * y = 0 \Rightarrow (x = 0 \lor y = 0)$$

Example 2. Assuming type parameters A and B, the following formulae are sound Event-B polymorphic theorems:

$$\forall a : \mathbb{P}(A), b : \mathbb{P}(A) \cdot a \subseteq b \Rightarrow (finite(b) \Rightarrow finite(a))$$
$$\forall f : A \leftrightarrow B, a : \mathbb{P}(A), b : \mathbb{P}(B) \cdot f \in a \nrightarrow b \Rightarrow (finite(a) \Rightarrow finite(f))$$

as it can be shown that the following four sequents are provable; hence satisfying the conditions stipulated in Definition 4.4.

$$\vdash_{\mathcal{D}} \quad \mathcal{D}(\forall a : \mathbb{P}(A), b : \mathbb{P}(A) \cdot a \subseteq b \Rightarrow (finite(b) \Rightarrow finite(a)))$$
$$\vdash_{\mathcal{D}} \quad \forall a : \mathbb{P}(A), b : \mathbb{P}(A) \cdot a \subseteq b \Rightarrow (finite(b) \Rightarrow finite(a))$$
$$\vdash_{\mathcal{D}} \quad \mathcal{D}(\forall f : A \leftrightarrow B, a : \mathbb{P}(A), b : \mathbb{P}(B) \cdot f \in a \nrightarrow b \Rightarrow (finite(a) \Rightarrow finite(f)))$$
$$\vdash_{\mathcal{D}} \quad \forall f : A \leftrightarrow B, a : \mathbb{P}(A), b : \mathbb{P}(B) \cdot f \in a \nrightarrow b \Rightarrow (finite(a) \Rightarrow finite(f))$$

Summary. In this section, we have shown how polymorphic theorems can be specified. We provided the sufficient proof obligations to ensure theorems are sound. We, also, demonstrated how theorems can be used in proofs when a suitable type substitution is provided. In §4.6, we show how certain polymorphic theorems can be used in a more pragmatic way as inference rules.

4.6 Inference Rules

In this section, we show how a special subset of polymorphic theorems can be manipulated in such a way that they can be used as inference rules. As mentioned earlier, polymorphic theorems achieve a two-fold objective. They can be used to ensure operator definitions capture the intended semantics. They can also be used in proofs as demonstrated in §4.5.3. We show that a polymorphic theorem with a specific structure can be used in a similar way to inference rules.

4.6.1 Defining Inference Rules

The following definition describes the syntactic properties satisfied by inference rules. As we develop this section, we will provide justifications for the different syntactic restrictions on inference rules.

Definition 4.6 (Event-B Inference Rule)**.** An Event-B inference rule is a pair $(\vec{G}, I)$ where:

1. I is an Event-B formula that is syntactically distinct from $\top$, called the infer clause,
2. $\vec{G}$ is a set of Event-B formulae, called the given clauses,
3. one of the following syntactic condition holds:

$$\mathcal{V}ar(I) \subseteq \bigcup_{H \in \vec{G}} \mathcal{V}ar(H)$$
$$\bigcup_{H \in \vec{G}} \mathcal{V}ar(H) \subseteq \mathcal{V}ar(I)$$

In the theory component, inference rules are defined according to Figure 4.6.

---

inference *name*
  [**automatic**] [**interactive**]
  vars $x_1, ..., x_n$
  condition $P(x_1, ..., x_n)$
  given $G_1, ..., G_m$
  infer I

Figure 4.6: Inference Rule Definition

The formula $P(x_1, ..., x_n)$ provides typing information for each of the variables occurring in the inference rule. The next subsection describes how an inference rule is validated in the theory component.

Intuitively speaking, inference rules are intended to be used in the following way. The infer clause of an inference rule may be matched against the goal of a sequent. If the matching succeeds, a backward proof step is achieved by making the (the instantiated) given clauses of the inference rule as the new sub-goals. Alternatively, the given clauses of an inference rule may be matched against the hypotheses of a sequent. If a suitable match is found for each given clause, a forward proof step is achieved by adding the (the instantiated) infer clause of the inference rule as a hypothesis.

Note that the infer clause has to be different from $\top$ since otherwise the inference cannot be of any use.[4] Moreover, the third condition in Definition 4.6 ensures that inference rules are at least applicable in one direction.

4.6.2 Using Inference Rules

Inference rules can be used in a backward style as well a forward style. If used in backward style, it discharges or splits the goal. If applied in a forward style, more hypotheses get generated. Let the following formula

$$\forall \vec{x} \cdot \vec{G} \Rightarrow I \tag{4.2}$$

be a sound polymorphic theorem. Let σ_t be a type substitution. Let σ be a variable substitution (as per §3.1.2) covering all the bound variables in $\forall \vec{x} \cdot \vec{G} \Rightarrow I$, i.e., $\vec{x}$. This is formally expressed as

$$\mathcal{D}om(\sigma) \;=\; \mathcal{V}ar(\vec{G} \Rightarrow I)$$

The combination of substitutions σ_t and σ provides instantiations for types and variables of the polymorphic theorem.

1. **Forward Inference.** Assume the following sequent whose provability is to be established:

   $$H, \sigma(\sigma_t(\vec{G})) \;\vdash_{\mathcal{D}}\; P \tag{4.3}$$

   where $\sigma(\sigma_t(\vec{G}))$ signifies a formula with a type and variable substitution applied to it. By introducing a suitable instance of the polymorphic theorem (4.2) as per rule $thm_{\mathcal{D}}$, we get the following sequent

   $$H, \sigma(\sigma_t(\vec{G})), \forall \vec{x} \cdot \sigma_t(\vec{G}) \Rightarrow \sigma_t(I) \;\vdash_{\mathcal{D}}\; P \tag{4.4}$$

   By applying rule $\forall hyp_{\mathcal{D}}$ on Sequent 4.4, we obtain the following sequents

   $$H, \sigma(\sigma_t(\vec{G})), \sigma(\sigma_t(\vec{G})) \Rightarrow \sigma(\sigma_t(I)) \;\vdash_{\mathcal{D}}\; P \tag{4.5}$$

   $$H, \sigma(\sigma_t(\vec{G})) \;\vdash_{\mathcal{D}}\; \bigwedge_{x \in \mathcal{D}om(\sigma)} \mathcal{D}(\sigma(x)) \;. \tag{4.6}$$

   By applying rule $\Rightarrow hyp_{\mathcal{D}}$ on Sequent 4.5, we obtain the following two sequents

   $$H, \sigma(\sigma_t(\vec{G})) \;\vdash_{\mathcal{D}}\; \sigma(\sigma_t(\vec{G})) \tag{4.7}$$

   $$H, \sigma(\sigma_t(\vec{G})), \sigma(\sigma_t(I)) \;\vdash_{\mathcal{D}}\; P \tag{4.8}$$

[4] Other formulae that can be shown by proof to be equivalent to $\top$, e.g., $\top \vee \bot$, are allowed. It is not always possible to perform syntactic checks to single out such formulae, so the liberty is left to the user to avoid them.

Sequent 4.7 can be discharged using the rule $hyp_{\mathcal{D}}$.

In summary, to prove the following sequent

$$H, \sigma(\sigma_t(\vec{G})) \ \vdash_{\mathcal{D}} \ P,$$

it suffices to show the provability of the following sequents

$$\begin{aligned} H, \sigma(\sigma_t(\vec{G})) \ &\vdash_{\mathcal{D}} \ \bigwedge_{x \in \mathcal{D}om(\sigma)} \mathcal{D}(\sigma(x)) \\ H, \sigma(\sigma_t(\vec{G})), \sigma(\sigma_t(I)) \ &\vdash_{\mathcal{D}} \ P \end{aligned}$$

2. **Backward Inference.** Assume the following sequent whose provability is to be established:

$$H \ \vdash_{\mathcal{D}} \ \sigma(\sigma_t(I)) \ , \tag{4.9}$$

where $\sigma(\sigma_t(I))$ signifies a formula with a type and variable substitution applied to it. By introducing a suitable instance of the polymorphic theorem (4.2) as per rule $thm_{\mathcal{D}}$, we get the following sequent

$$H, \forall \vec{x} \cdot \sigma_t(\vec{G}) \Rightarrow \sigma_t(I) \ \vdash_{\mathcal{D}} \ \sigma(\sigma_t(I)) \ . \tag{4.10}$$

By applying rule $\forall hyp_{\mathcal{D}}$ on Sequent 4.10, we obtain the following sequents

$$H, \sigma(\sigma_t(\vec{G})) \Rightarrow \sigma(\sigma_t(I)) \ \vdash_{\mathcal{D}} \ \sigma(\sigma_t(I)) \tag{4.11}$$

$$H \ \vdash_{\mathcal{D}} \ \bigwedge_{x \in \mathcal{D}om(\sigma)} \mathcal{D}(\sigma(x)) \ . \tag{4.12}$$

By applying rule $\Rightarrow hyp_{\mathcal{D}}$ on Sequent 4.11, we obtain the following two sequents

$$H \ \vdash_{\mathcal{D}} \ \sigma(\sigma_t(\vec{G})) \tag{4.13}$$

$$H, \sigma(\sigma_t(I)) \ \vdash_{\mathcal{D}} \ \sigma(\sigma_t(I)) \ . \tag{4.14}$$

Sequent 4.14 can be discharged using the rule $hyp_{\mathcal{D}}$.

In summary, to prove the following sequent

$$H \ \vdash_{\mathcal{D}} \ \sigma(\sigma_t(I)) \ ,$$

it suffices to show the provability of the following sequents

$$\begin{aligned} H \ &\vdash_{\mathcal{D}} \ \bigwedge_{x \in \mathcal{D}om(\sigma)} \mathcal{D}(\sigma(x)) \\ H \ &\vdash_{\mathcal{D}} \ \sigma(\sigma_t(\vec{G})) \ . \end{aligned}$$

The previous development (forward and backward inference) is carried out by importing and appropriately instantiating a theorem. However, the use of theorems as inference rules can be automated to a certain degree. This is where condition (3) of Definition 4.6 comes into play. We have the following two definitions with regard to inference rule applicability.

Definition 4.7 (Forward-applicable Event-B Inference Rule)**.** An Event-B inference rule $(\vec{G}, I)$ is said to be forward-applicable if the following condition holds:

$$\mathcal{V}ar(I) \subseteq \bigcup_{H \in \vec{G}} \mathcal{V}ar(H)$$

The intuition behind Definition 4.7 is that an inference rule can be applied in a forward fashion if its given clauses contains all variables of the inference rule. This means one-way matching can be used to find a binding that unifies some hypotheses with the given clauses. Since the binding will have mappings for all variables, the infer clause can be instantiated using that same binding.

Definition 4.8 (Backward-applicable Event-B Inference Rule)**.** An Event-B inference rule $(\vec{G}, I)$ is said to be backward-applicable if the following condition holds:

$$\bigcup_{H \in \vec{G}} \mathcal{V}ar(H) \subseteq \mathcal{V}ar(I)$$

The intuition behind Definition 4.8 is that an inference rule can be applied in a backward fashion if its infer clause contains all variables of the inference rule. This means matching can be used to find a binding that unifies the goal with the infer clause. Since the binding will have mappings for all variables, all given clauses can be instantiated using that same binding.

We, now, summarise the two possible ways in which inference rules can be applied. Consider the sound Event-B inference rule $(\vec{G}, I)$. If the rule is forward-applicable, it can be applied according to the following proof tree

$$\frac{H, \sigma(\sigma_t(\vec{G})) \vdash_{\mathcal{D}} \bigwedge_{x \in \mathcal{D}om(\sigma)} \mathcal{D}(\sigma(x)) \qquad H, \sigma(\sigma_t(\vec{G})), \sigma(\sigma_t(I)) \vdash_{\mathcal{D}} P}{H, \sigma(\sigma_t(\vec{G})) \vdash_{\mathcal{D}} P} \; forInf_{\mathcal{D}}$$

where σ_t and σ are suitable substitutions. If the rule is backward-applicable, it can be applied according to the following proof tree

$$\frac{H \vdash_{\mathcal{D}} \bigwedge_{x \in \mathcal{D}om(\sigma)} \mathcal{D}(\sigma(x)) \qquad H \vdash_{\mathcal{D}} \sigma(\sigma_t(\vec{G}))}{H \;\vdash_{\mathcal{D}}\; \sigma(\sigma_t(I))} \; backInf_{\mathcal{D}}$$

where σ_t and σ are suitable substitutions.

4.6.3 Validating Inference Rules

As mentioned in the previous section, inference rules as defined above are a special case of implicative polymorphic theorems. For each inference rule, we can derive the appropriate polymorphic theorem.

Definition 4.9 (Derived Theorem)**.** The following formula is called the derived theorem of the Event-B inference rule $(\overrightarrow{G}, I)$:

$$\forall \vec{x} \cdot \vec{G} \Rightarrow I$$

where $\vec{x}$ are the free variables in all of $\vec{G}$ and I.

The following definition describes the sufficient conditions under which an inference rule is considered valid.

Definition 4.10 (Sound Inference Rule)**.** An inference rule $(\vec{G}, I)$ is said to be sound if its derived polymorphic theorem is sound.

Summary. In this section, we have demonstrated a pragmatic approach to using polymorphic theorems. Inference rules are intended to relieve the user from explicitly providing instantiations for type and ordinary variables of a polymorphic theorem. We have shown how inference rules can be used in backward proof, by splitting the goal, and forward proof, by generating new hypotheses. We concluded our discussion by outlining the proof obligations to ensure soundness of user-defined inference rules.

4.7 Polymorphic Operators

A new Event-B polymorphic operator can be defined in a theory by providing the following information:

1. *Parser Information*: this includes the syntax, the notation (infix or prefix), and the syntactic class (term or formula). The precedence of the operator is not provided by the user.
2. *Type Checker Information*: this includes the types of the child arguments, and the resultant type if the operator is a term operator.
3. *Prover Information*: this includes the well-definedness of the operator as well as its definition which may be used to reason about it.

operator *syntax* [**commutative**] [**associative**]
(**prefix** | **infix**)
args $x_1 \in T_{x_1}, ..., x_n \in T_{x_n}$
condition $P(x_1, ..., x_n)$
definition $Q(x_1, ..., x_n)$

Figure 4.7: Operator Definition

Figure 4.7 describes the general structure of a new operator definition, where:

1. ***'syntax'***: defines the syntax of the new operator. It has be distinct from previously used operator syntaxes as our approach does not allow operator overloading.
2. **'prefix'** or **'infix'**: defines the type of the notation that will be used for this operator either infix (e.g., $a\ op\ b$) or prefix (e.g., $op(a, b)$). At the time of writing this thesis, postfix operators were not supported.
3. **'commutative'**: indicates whether the operator is commutative. This particular property of operators triggers the generation of a proof obligation.
4. **'associative'**: indicates whether the operator is associative. This particular property of operators triggers the generation of a proof obligation.
5. **'args'**: defines the arguments of the operator. Each argument must have a name and a type. Names of the arguments are pairwise distinct.
6. **'condition'**: provides the well-definedness condition to be generated for this operator. We will show later how concrete well-definedness conditions are correctly generated from the above definition.
7. **'definition'**: provides the direct definition of the operator in terms of the existing mathematical language. The syntactic class of the operator is inferred from the syntactic class of $Q(x_1, ..., x_n)$. If $Q(x_1, ..., x_n)$ is a term, then the resultant type of the operator is the type of $Q(x_1, ..., x_n)$.

4.7.1 Example: The Sequence Operator

A sequence is an ordered list of objects where the same object can occur multiple times at different positions. It is, therefore, easy to see that a sequence can be defined as a polymorphic operator. The following snippet provides a definition of a sequence in Event-B.

theory *SeqThy*
type parameters S

operator *Seq*
(**prefix**)
args $a \in \mathbb{P}(S)$
condition $\top$
definition $\{f, n \cdot f \in 1..n \rightarrow a \mid f\}$

In the above snippet, $1..n$ denotes a contiguous integer range. The previous definition describes the set of all sequences of the set a; each sequence is defined as a total function from an integer range to the set a. The following typing rule[5] is generated for the operator *Seq*:

$$\frac{\mathbf{type}(a) = \mathbb{P}(S)}{\mathbf{type}(Seq(a)) = \mathbb{P}(\mathbb{P}(\mathbb{Z} \times S))}.$$

In the following snippet, the formula operator *EmptySeq* takes a sequence, and *'returns'* whether the sequence is empty. The term operators $HeadSeq$ and $TailSeq$ calculate the head and the tail of a non-empty sequence respectively.

operator *EmptySeq*
(**prefix**)
args $s \in \mathbb{P}(\mathbb{Z} \times S)$
condition $s \in Seq(S)$
definition $card(s) = 0$

operator *HeadSeq*
(**prefix**)
args $s \in \mathbb{P}(\mathbb{Z} \times S)$
condition $\neg EmptySeq(s)$
definition $s(1)$

operator *TailSeq*
(**prefix**)
args $s \in \mathbb{P}(\mathbb{Z} \times S)$
condition $\neg EmptySeq(s)$
definition $\{i \cdot i \in 1..(card(s) - 1) \mid i \mapsto s(i + 1)\}$

The following typing rule is generated for the *EmptySeq* formula operator:

$$\mathbf{type}(s) = \mathbb{P}(\mathbb{Z} \times S)$$

[5]Note that the type of an individual sequence is $\mathbb{P}(\mathbb{Z} \times S)$, i.e., a set of pairs. Therefore, the type of a set of sequences is $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times S))$

The following typing rules are generated for the head and tail operators:

$$\frac{\mathbf{type}(s) = \mathbb{P}(\mathbb{Z} \times S)}{\mathbf{type}(HeadSeq(s)) = S}$$

$$\frac{\mathbf{type}(s) = \mathbb{P}(\mathbb{Z} \times S)}{\mathbf{type}(TailSeq(s)) = \mathbb{P}(\mathbb{Z} \times S)}$$

4.7.2 Operator Properties

In this section, we describe the different aspects of a new operator definition. More specifically, we focus on well-definedness, associativity and commutativity.

4.7.2.1 Well-Definedness

An important aspect of defining an operator is the well-definedness condition to be used. A simple strategy may use the well-definedness of the operator's direct definition. An advantage of a user-supplied condition is the possibility of strengthening well-definedness conditions to simplify proofs. In order to ensure that a supplied condition is in fact stronger than the default (i.e., the one inferred from the direct definition), proof obligations are generated.

As discussed in §2.4.3, an important property of well-definedness conditions is that they are themselves well-defined, i.e.,:

$$\mathcal{D}(\mathcal{D}(P)) \;\Leftrightarrow\; \top \quad \text{for any formula or term } P$$

There is a possibility that the supplied well-definedness condition may not, in some cases, be well-defined (e.g., $HeadSeq$ well-definedness condition). Therefore, the complete well-definedness condition of an operator is the following:

$$\mathcal{D}(P(x_1, ..., x_n)) \wedge P(x_1, ..., x_n)$$

As an example, the default well-definedness condition of the $HeadSeq$ (and, coincidently, $TailSeq$) operator is

$$s \in Seq(S) \wedge \neg EmptySeq(s)$$

To ensure that the supplied well-definedness condition is stronger than the default one, the following proof obligation is generated:

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T_{x_1}, ..., x_n \in T_{x_n} \cdot (\mathcal{D}(P(x_1, ..., x_n)) \wedge P(x_1, ..., x_n)) \Rightarrow \mathcal{D}(Q(x_1, ..., x_n))}$$

The well-definedness strength proof obligation is justified in §5.1 of [102].

As an example, the following proof obligations are generated for $EmptySeq$ and $HeadSeq$, respectively:

$$\vdash_{\mathcal{D}} \forall s \in \mathbb{P}(\mathbb{Z} \times S) \cdot s \in Seq(S) \Rightarrow finite(s)$$
$$\vdash_{\mathcal{D}} \forall s \in \mathbb{P}(\mathbb{Z} \times S) \cdot (s \in Seq(S) \land \neg EmptySeq(s)) \Rightarrow (s \in \mathbb{Z} \pfun S \land 1 \in dom(s))$$

using the following expansions:

$$\begin{array}{rcl} \mathcal{D}(card(s)) & \widehat{=} & finite(s) \\ \mathcal{D}(s(1)) & \widehat{=} & s \in \mathbb{Z} \pfun S \land 1 \in dom(s) \end{array}$$

4.7.2.2 Commutativity

An operator is said to be commutative if it is a binary operator whose arguments are of the same type, and the following formula is valid:

$$\begin{array}{lcll} Q(x_1, x_2) & = & Q(x_2, x_1) & \text{if the operator is a term operator, or} \\ Q(x_1, x_2) & \Leftrightarrow & Q(x_2, x_1) & \text{if the operator is a formula operator.} \end{array}$$

Example. Consider the definition of the AND boolean operator:

operator AND **commutative associative**
(**infix**)
args $b_1 \in BOOL,\ b_2 \in BOOL$
condition $\top$
definition $bool(b_1 = TRUE \land b_2 = TRUE)$

Note that $BOOL$ is a built-in type in Event-B; it contains the values $TRUE$ and $FALSE$. The operator *bool* is a built-in operator that takes a predicate as an argument; its resultant type is $BOOL$. In this case, the following formula describes the condition which asserts that the AND operator is commutative:

$$\forall b_1 \in BOOL, b_2 \in BOOL \cdot bool(b_1 = TRUE \land b_2 = TRUE) = bool(b_2 = TRUE \land b_1 = TRUE)$$

More generally, if an operator is defined by the user to be commutative, then

1. if the operator is a term operator, the following proof obligation is generated

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T, x_2 \in T \cdot Q(x_1, x_2) = Q(x_2, x_1)}$$

2. if the operator is a formula operator, the following proof obligation is generated

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T, x_2 \in T \cdot Q(x_1, x_2) \Leftrightarrow Q(x_2, x_1)}$$

4.7.2.3 Associativity

A term operator is said to be associative if:

- it is a binary operator whose arguments are of the same type,
- the resultant type of the operator is the same as that of the arguments,
- the following formula is valid:

$$Q(Q(x_1, x_2), z) = Q(x_1, Q(x_2, z)) \ .$$

If a term operator is defined by the user to be associative, the following proof obligation is generated:

$$\boxed{\vdash_{\mathcal{D}} \forall x_1 \in T, x_2 \in T, z \in T \cdot Q(Q(x_1, x_2), z) = Q(x_1, Q(x_2, z))}$$

Summary. In this section, we have shown how polymorphic operators can be defined in the theory construct. We discussed important aspects of operators, including syntax, well-definedness, commutativity and associativity. We also provided the different proof obligations that are necessary to validate user-defined polymorphic operators.

4.8 Datatypes

Datatypes are important ingredients of many formalisms and programming languages [107, 75]. In this section, we will show by means of simple examples how the theory component can be used to define new datatypes. In our discussion, we do not provide a rigorous treatment of the subject, nor do we claim that the development has reached a mature stage. However, as pointed out in §5 [102], datatypes can be added on top of the logic of Event-B as defined by Schmalz. The syntactic restrictions placed on datatypes resemble those placed on Isabelle/HOL datatypes as developed in [27].

In this brief treatment, we will be concerned by datatypes which are generated from a number of constructors. Each element of the type can be written as a constructor term. Moreover, the datatypes are freely generated which requires the constructors to be distinct and injective. This ensures that every element of the newly-defined datatype is denoted by a unique constructor term, and consequently, a structural induction theorem holds for such datatype. The structural induction theorem enables the definition of operators by primitive recursion [27, 107].

A new datatype is introduced by providing the following:

1. A type constructor operator,
2. A number of element constructors one of which must be a base constructor,
3. Extensionality axioms to ensure constructed elements are uniquely determined by their constituents,
4. Disjointness axioms ensuring that distinct constructors yield distinct elements,

5. An induction axiom.

Generally speaking, a datatype specification in the theory construct has the following form:

$$t(\alpha_1, ..., \alpha_n) \quad ::= \quad C_1(\pi_1^1, ..., \pi_1^m) \quad | \quad ... \quad | \quad C_k(\pi_k^1, ..., \pi_k^l)$$

where α_1,..., α_n are type parameters, C_1,..., C_k are the constructors of the new datatype, and each of π_i^j is a type that may only refer to the type parameters of the datatype. Constructor names must be distinct. Types in Event-B are assumed to be non-empty, and this must hold for datatypes. As such, each newly defined datatype must have a base constructor, i.e., a constructor that does not refer to the datatype being defined. Furthermore, the admissibility check discussed in [27] has to be enforced to avoid a major issue with nesting of datatype definitions. If the admissibility check is dropped, the datatype cannot be constructed [27].

In the context of Event-B, the admissibility check rules out the following datatype definition

$$t(\alpha) \quad ::= \quad C_1 \quad | \quad C_2(\mathbb{P}(t))$$

since there is no injective function of type $\mathbb{P}(t) \to t$ by Cantor's theorem.

4.8.1 A List Datatype

As an example, we consider the definition of a list datatype using the following syntactic sugar:

$$List(\alpha) \ ::= \ nil | \ cons(\alpha, \ List(\alpha))$$

where α is a type parameter. In this case, we have the following:

1. type constructor operator: $List(\alpha)$,
2. element constructors: nil and $cons$,
3. extensionality axioms:

$$\forall x, x', l, l' \cdot cons(x, l) = cons(x', l') \Rightarrow x = x' \wedge l = l'$$

4. disjointness axioms:

$$\forall x, l \cdot cons(x, l) \neq nil$$

5. induction axiom:

$$P(nil) \wedge (\forall x, l \cdot P(l) \Rightarrow P(cons(x, l))) \Rightarrow (\forall l \cdot P(l))$$

The theory component allows the definition and use of datatype accessors. The list datatype definition can be more succinctly written as:

$$\begin{array}{lll} List(\alpha) & ::= & nil \\ & | & cons(head : \alpha, \ tail : List(\alpha)) \ . \end{array}$$

The previous definition introduces the a number of expressions to the Event-B mathematical language. $List(\alpha)$ is a type expression as well as a set expression. nil is an expression of type $List(\alpha)$.$cons(x, l)$ is an expression of type $List(\alpha)$. $head$ is a partial operator of type $List(\alpha) \pfun \alpha$, and $tail$ is a partial operator of type $List(\alpha) \pfun List(\alpha)$. The $head$ and $tail$ are operators whose well-definedness conditions are the following:

$$\begin{array}{rcl} \mathcal{D}(head(l)) & \widehat{=} & \exists x, l_0 \cdot l = cons(x, l_0) \\ \mathcal{D}(tail(l)) & \widehat{=} & \exists x, l_0 \cdot l = cons(x, l_0) \end{array}$$

Pattern-based recursive operators can be specified by providing definitions corresponding to each constructor of the concerned datatype. The size of a list can be defined by means of the following operator:

operator *listSize*
(**prefix**)
args $l \in List(T)$
definition
case l
$listSize(nil) = 0$
$listSize(cons(x_0, l_0)) = 1 + listSize(l_0)$

Prior to wour work, only built-in types and carrier sets can be used in models. Datatypes can be constructed axiomatically in contexts by defining a carrier set (corresponding to the datatype) and a number of injective functions to specify the datatype constructors. However, this approach has two drawbacks. Firstly, the datatype is not polymorphic as it uses carrier sets. Secondly, it uses contexts for a purpose for which they were not initially intended as discussed in §4.1.1. We argue that datatypes in the theory construct address the aforementioned drawbacks.

Summary. In this section, we briefly presented how datatypes are specified in the theory construct. The particular issue of datatype admissibility is highlighted. The objective of this section was to provide a cursory overview of datatypes in theories. The work on datatypes in the logic of Event-B is not complete, and it could further be complemented by adding facilities for mutually recursive datatypes.

4.9 Related Work

The related work is divided into four sub-sections corresponding to the different contributions of this chapter.

4.9.1 Module Systems in Specification Languages

Modularity is an important concern in specification and programming languages. Modern programming languages such as Java and C++ incorporate difference constructs to provide a modular approach to software development, e.g., classes and inheritance. Maude is a reflective language

and system supporting both equational and rewriting logic specification and programming for a wide range of applications [35, 36]. Rewriting logic is a logic of concurrent change that can naturally deal with state and with concurrent computations [79]. Maude provides a modular system for specifying rewrite theories. Each module provides sorts, kinds and operators, and can have equations, memberships and rules [35]. The theory construct is similar to a module in Maude given the facilities provided for specifying operators, types and rewrite rules. However, theory development is secondary to model development, i.e., contexts and machines, in Event-B. Theories should not be considered as modelling elements in a specification. Rather, their role remains as a meta-reasoning vehicle for the logic of Event-B rather than the specification language of Event-B, i.e., outer syntax. A similar comparison can be drawn between Event-B theories and OBJ3 [55] modules.

Extended ML is a framework for specification and formal development of Standard ML (SML) programs. Developing a program in Extended ML means writing a specification of a generic SML module and then refining this specification in a top-down fashion by means of a number of refinement steps until an SML program is obtained [100, 71]. The counterpart of an Extended ML module is in fact a machine. However, parallels can be drawn between a module and a theory. A theory can be used to specify operators, types and proof rules in a modular fashion. Hierarchies of theories exist to specify a collection of related mathematical structures. However, a key difference between theories and modules in Extended ML is that code generation is not a requirement for theories. In fact, code generation is more pressing in the case of contexts and machines. As such, we conclude that more parallels can be drawn between Event-B models and Extended ML modules than between Event-B theories and Extended ML modules.

Isabelle [89] and PVS [92] theories are similar to Event-B theories, but are wider in scope. Theories in Isabelle and PVS can be used to carry significant modelling and reasoning activities. We argue that combining modelling and theory development in Event-B provides a comparable level of sophistication to that of Isabelle and PVS theories. Event-B modelling uses set theory which can provide powerful expressive power that is close to higher order logic [13]. The addition of the theory component ensures that polymorphism can be exploited to enhance the expressive power of the Event-B mathematical language.

4.9.2 Prover Extensibility

The architecture of proof tools continues to stir up much heated debate. One of the main talking points is how to strike a reasonable balance between three important attributes of the prover: efficiency, extensibility and soundness. In [62], Harrison outlines three options to achieve prover extensibility:

1. If a new rule is considered to be useful, simply extend the basic primitives of the prover to include it.
2. Use a full programming language to specify new rules using the basic primitives. The new rules ultimately decompose to these primitives.
3. Incorporate the *reflection* principle, so that the user can add and verify new rules within the existing infrastructure.

Many theorem provers including Isabelle [89] and HOL [58] employ the LCF approach. The functional language ML [88] is used to implement these systems, and acts as their meta-language. The approach taken by such systems is to use ML to define data types corresponding to logical entities such as terms and theorems. A number of ML functions are provided that can generate theorems; these functions implement the basic inference rules of the logic. The ML type system ensures that theorems are only constructed by the aforementioned functions. Therefore, the LCF approach offers both "reliability" and "controllability" of a low level proof checker combined with the power and flexibility of a sophisticated prover [62]. On the flip side, however, a major drawback for this approach is that each newly developed proof procedure must decompose into the basic inference rules. There are cases where this may not be possible or indeed an efficient solution e.g., the truth table method for propositional logic [40].

The PVS [92] system follows a similar approach to LCF with more liberal support for adding external provers. This liberality comes at a risk of encountering soundness bugs. It, however, presents the user with several choices of automated provers which may ease the proving experience. A comparison between Isabelle/HOL and PVS from a user's point of view is presented in [59]. Interestingly, it mentions that "soundness bugs are hardly ever unintentionally explored" during proof, and that "most mistakes in a system to be verified are detected in the process of making a formal specification". A similar experience is reported when using the Rodin platform [82].

The Mural formal development system [73] consists of a VDM support tool and a proof assistant. However, in essence, it provides support for many-sorted predicate calculi which are expressible in natural deduction style. The Mural system allows adding internally proved rules i.e., rules that follow directly from existing rules. This results in the exclusion of a large class of rules that could be proved by employing a "more sophisticated meta-reasoning". Adding new rules in Mural can be achieved through extending existing theories providing a verifiably "open system".

Programming tools such as JML [30], ESC/Java [38], Boogie [22] (Spec# [23] program verifier) and VCC [37] provide capabilities to verify computer programs. Verification conditions are generated, and passed on to external provers, e.g., SMT solvers. Since theorem proving is not an integrated component in these tools, prover extensibility is not an immediate concern. However, the choice of highly configurable and customisable tools is readily available, e.g., Isabelle and SMT solvers. Note that a similar approach is adopted by VDM [70].

The KIV [21] theorem prover is a tool for formal development and interactive verification. KIV provides proof support for all elements of the specification language based on sequent calculus, rewriting and symbolic execution of programs. This theorem prover follows a tactic-based approach to proof, and provides a number of proof heuristics that can only be modified or augmented by the system developer. Facilities are not provided for specifying new proof procedures by system users. This particular limitation of KIV is similar to the limitations of the Event-B toolset prior to our work.

The KeY System [17, 18] is a formal software development toolset which proposes the integration of design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. Taclets [53] provide a mechanism by which proof rules can be defined for the KeY System. For example, a very simple taclet could be written as follows

```
find(b -> c ==>) if (b ==>) replacewith (c ==>) heuristics(simplify)
```

Listing 4.1: Simple Taclet

This taclet indicates that an implication $b \Rightarrow c$ should be replaced by formula c if b can be found in the left hand side of sequent (i.e., hypotheses). The aforementioned taclet is part of proof heuristics called 'simplify'. Even though, we do not explicitly support proof directives (such as 'find'), there is a limited implicit support for such constructs as exemplified in single conditional rewrite rule application and inference rules.

Our approach does not necessarily subdue the old mechanism of extending the prover. As such, the new prover architecture resembles that of PVS. It still allows the liberality of integrating external decision procedures (e.g., for arithmetic) while providing a collection of sound rules. On the other hand, verifying the soundness of added rules using proof obligations enables meta-reasoning within the same platform. This can be viewed as a limited incorporation of the reflection principle within Rodin. The limitations of our approach, however, are similar to the limitations of the Mural architecture, since sophisticated meta-reasoning is not possible at the moment.[6]

4.9.3 Language Extensibility

Language extensibility is a major concern in formal methodologies. Isabelle/HOL achieves a good level of extensibility through polymorphism. It also benefits from the availability of a meta-logic that can be used to specify operators with good control over syntactic representations [93]. The generic nature of Isabelle enables the specification of many logics, and, suitably, there is an attempt to encode Event-B in Isabelle.

Language extensibility is a real concern in PVS as discussed in [76]. PVS allows the use of parametrised theories which offers some of the benefits of language extensibility such as reusability. Both PVS and Isabelle/HOL provide facilities to define and use datatypes and recursive definitions. In both formalisms, when a new datatype is defined, a simple theory containing at least an induction principle is provided.

4.9.4 Datatypes

Datatypes are an important ingredient in specification and programming languages. Abstract datatypes play a major role in programming language such as Java, C++ and the functional language ML. Algebraic datatypes describe the theory behind the creation of types and the operations that manipulate and create elements of the said types. Note that abstract datatypes can be modelled using contexts and machines , as is carried out in [99]. However, such models of datatypes do not make the specified datatype available as a type for the subsequent specifications. This is in contrast with the specification languages Maude [35] and OBJ3 [55], where most types

[6] Predicate variables have been added to the mathematical language since Rodin 2.0. However, meta-reasoning in Rodin can be substantially enhanced by adding support for the well-definedness operator as a syntactic extension to the mathematical language.

are constructed algebraically, and made available for reuse as types. The objective of datatypes in theories is to provide further types (beside the built-in types and carrier sets) that can be used in modelling, e.g., for data refinement.

Datatypes can be defined in Isabelle/HOL [89] and PVS [92]; in both formalism, a theory is readily available to reason about the created datatype. In particular, [27, 107] provide an overview of the construction of datatypes in Isabelle/HOL. And as pointed out by Schmalz [102], the construction of datatypes could follow a similar path in the logic of Event-B. Note that this construction is absent in this thesis, as datatypes in the theory construct is a case of 'practice preceding theory'. Nonetheless, it provides a starting point for further research on the logic of Event-B and its possible extensions.

4.10 Summary

In this chapter, we presented an approach that improves the extensibility of the Event-B language and prover. The theory construct is used to define and validate rewrite rules as well as polymorphic theorems. Proof obligations are generated to ensure that soundness is maintained. We have shown how the theory construct can be used to specify rewrite rules in order to enhance the rewriting capabilities of Rodin. The justification for the work on rewriting in Event-B is presented in Chapter 3. Next, we introduced polymorphic theorems and presented how they can be incorporated in proofs. Inference rules provide a convenient mechanism by which user can apply certain polymorphic theorems. Furthermore, we addressed language extensibility issues by describing how polymorphic operators can be specified in the theory construct. The logical foundation behind support for polymorphic operators can be found in [102]. A minor contribution of our work is the addition of support for datatypes in the Event-B mathematical language. In summary, the work in this chapter has its theoretical foundation in Chapter 3 and [102], and resulted in providing effective tool support for extending the Rodin proving infrastructure as will be shown in Chapter 5 and 6.

Chapter 5

Tool Support: Theory Plug-in

It is widely accepted that formal methods are becoming more essential to software development [10, 31, 110]. There exist several cases that show the applicability and usefulness of formal techniques in software engineering [39]. An important component of any successful formal methodology is tool support. Effective tool support facilitates the integration of formal methods into the development process of computer systems [9]. It can be even argued that tool support is the most important factor in determining the success or otherwise of any formal method.

Isabelle [95] boasts an effective set-up that combines soundness and usability. It also provides a powerful mechanism for embedding logics. One of the most attractive attributes that contributed to the success of Isabelle is the LCF architecture as discussed in §4.9. PVS [91] provides a theorem prover consisting of a variety of primitive inference procedures. PVS employs Gnu or X Emacs to provide an integrated environment for its language and prover. One of the many strengths of PVS is the possibility of integrating external decision procedures (e.g., for arithmetic). The Rodin platform provides an extensible toolset for developing and reasoning about Event-B models. Rodin includes a collection of tools that are necessary for a reactive development environment. In this chapter, we shed some light on some of the important features of Rodin.

This chapter is structured in the following way. The theory component is introduced together with the appropriate tooling. Theory deployment is described in practical terms. We conclude by describing how the different mathematical and prover extensions can be used in models and proofs. Our aim in this chapter is to show how the different ideas presented in Chapter 4 have been implemented with the objective of addressing the extensibility issues outlined in §1.1. We undertook the development of the Theory plug-in as part of our research; it started as a proof of concept, and evolved to a solid platform for reasoning about Event-B extensions.

5.1 The Theory Plug-in

The Theory plug-in embodies many of the ideas presented in this thesis. It is our solution to the different extensibility issues described in §1.1. The Theory plug-in benefits from the highly configurable and extensible nature of the Rodin platform in the following aspects:

1. **The Rodin Database**. The Theory plug-in contributes the theory component as a Rodin file.
2. **Rodin Tooling**. The Theory plug-in provides a static checker and proof obligation generator for theory files.
3. **Dynamic AST**. The Theory plug-in provides a front-end to the Rodin dynamic parser for the mathematical language.
4. **Reasoners and Tactics**. The Theory plug-in dynamically creates reasoners and tactics as wrappers around user-specified proof rules.

The Theory plug-in follows the Rodin philosophy by:

1. adopting the familiar approach of reactive development, and
2. using proof obligations to ensure soundness preservation.

5.1.1 The Theory Construct

The theory construct (component) is a Rodin file acting as a place holder for mathematical and proof extensions. The theory construct can be used to specify:

1. mathematical extensions including datatypes and operators with direct or primitive recursive definitions, and
2. proof extensions including polymorphic theorems, rewrite and inference rules.

Theories have the structure described in Figure 4.3. A theory is parametrised by means of a number of type parameters. All extensions are polymorphic on the type parameters to which they refer.

A new theory can be created by specifying its name and its hosting project as per Figure 5.1.

Event-B theories can include a number of the following elements:

1. **Theory Imports**. This specifies a directed relationship between the parent theory (the importer) and the referenced theory (the importee). The importer theory can refer and use any of the extensions defined in the importee theory. The import relationship enables the importing theory to use all mathematical and proof extensions defined in the imported theory. The import directive enables the creation of theory hierarchies. For instance, two separate theories can be created to define sequences and inductive lists, and a third theory importing the previous two theories can be created to specify an isomorphism between sequences and inductive lists. In effect, the import directive establishes a partial order on the collection of theories within a project. The imported theories need not be instantiated with type parameters a la PVS [91].
2. **Type Parameters**. This defines the types on which theory extensions may be polymorphic. Type parameters are similar to carrier sets in contexts; the only assumption regarding type parameters is non-emptiness.

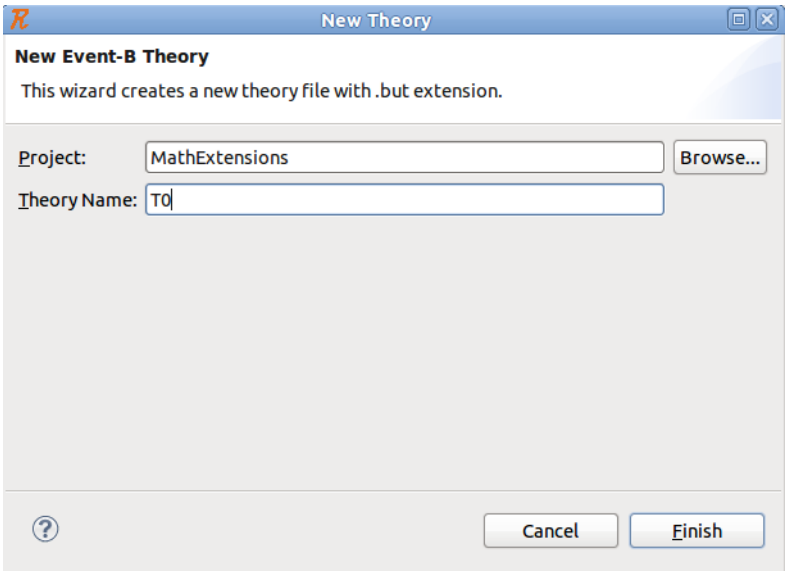


Figure 5.1: Creating a New Theory

3. **Datatypes**. Datatypes are defined by providing the following information:
   (a) the type expression syntax e.g., *List*,
   (b) the type parameters of the datatype e.g., a single type parameter T for *List*,
   (c) a number of element constructors e.g., *nil* and *cons* for *List*. Each constructor may have a number of destructors (accessors) e.g., *head* and *tail* accessors for *cons*.

   Figure 5.2 shows a definition of inductive lists.
4. **Operators**. Operators are defined by providing the following information:
   (a) the syntax symbol of the operator,
   (b) the syntactic class (i.e., predicate or expression),
   (c) the notation (only prefix and infix are currently supported),
   (d) the list of arguments and their types,
   (e) the condition under which the operator is to be used,
   (f) a definition which can be 1) direct, or 2) primitive recursive.

   Figure 5.3 illustrates a direct definition for the sequence operator. Figure 5.4, on the other hand, illustrates a primitive recursive definition for the list size operator.
5. **Theorems**. A polymorphic theorem can be added by specifying its name (i.e., its identifier) and its formula. Figure 5.5 shows an simple theorem about the finiteness of sequences.
6. **Proof Rules**. Two types of proof rules can be defined: rewrite and inference rules. Meta-variables are used as variable patterns in rules to facilitate 1) pattern matching, and 2) type inference and checking. A meta-variable (as shown in Figure 5.6) has a name and a type.
   (a) **Rewrite Rules**. A rewrite rule can be defined by providing the following information:

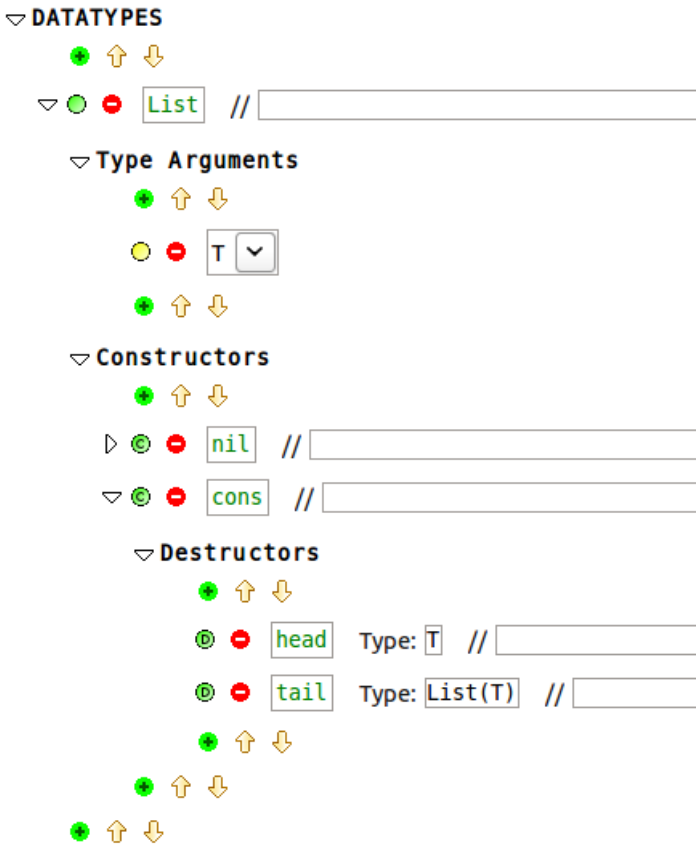


Figure 5.2: Definition of Inductive Lists

i. the left hand side to be rewritten,

ii. the applicability of the rule (i.e., automatic, interactive or both),

iii. the description of the rule for user interface purposes, and

iv. the right hand sides to which the left hand side can be rewritten; each right hand side is guarded by a condition.

Figure 5.7 shows a simple rewrite rule.

(b) **Inference Rules**. An inference rule can be defined by providing the following information:

i. the applicability of the rule (i.e., automatic, interactive or both),

ii. the description of the rule for user interface purposes,

iii. the given clauses of the inference rule, and

iv. the infer clause of the inference rule.

Figure 5.8 shows an example inference rule.

5.1.2 Theory Static Checking

Event-B theories are subject to static checking. The theory static checker inspects unchecked theory files (with file extension '.tuf'), and produces checked theory files (with file extension

Figure 5.3: Operator with a Direct Definition

Figure 5.4: Operator with a Primitive Recursive Definition

'.tcf'). The following non-exhaustive list enumerates the checks implemented in the Theory plug-in:

1. **Import Checks**.
   - *Non-circularity of import relationship*,
   - *Redundancy of import relationship*: in case a theory is imported more than once; this is particularly useful if a theory is imported directly (using an import directive) and indirectly (by virtue of the transitivity of the import directive).
2. **Datatype Checks**.
   - *Syntax symbols clash*: for the type expression as well as constructors and accessors,

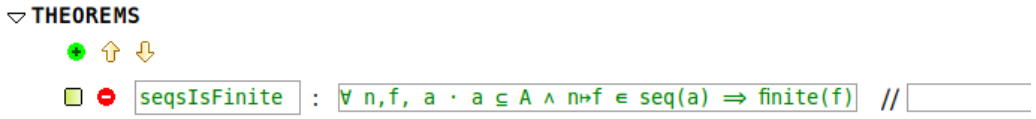


Figure 5.5: A Polymorphic Theorem

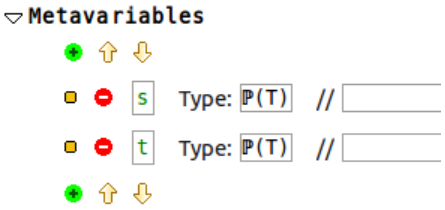


Figure 5.6: Meta-variables

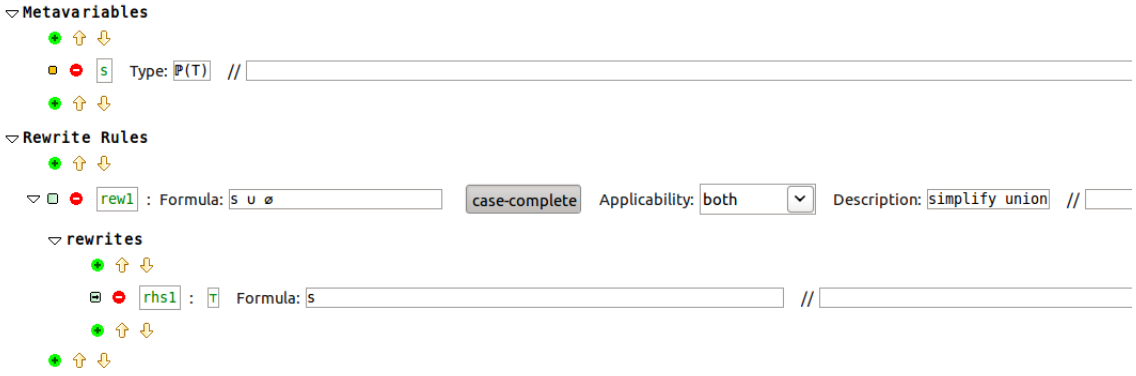


Figure 5.7: A Rewrite Rule

- *Presence of a base constructor*: each datatype definition must include a base constructor, and
- *Admissibility check*: see §4.8 for more on this check.

3. **Operator Checks**.
   - *Syntax symbols clash*: for the operator syntax symbol,
   - *Parsing and typing of operator arguments*,
   - *Parsing and type checking of well-definedness conditions*,
   - *Parsing and type checking of direct definitions*,
   - *Uniqueness of definition*: only one definition is allowed for each operator,
   - *Constructor coverage*: for primitive recursive definitions,
   - *Operator properties checks*: for example, an operator with a single argument cannot be tagged associative or commutative.
4. **Theorem Checks**.
   - *Parsing and type checking of the formula*,

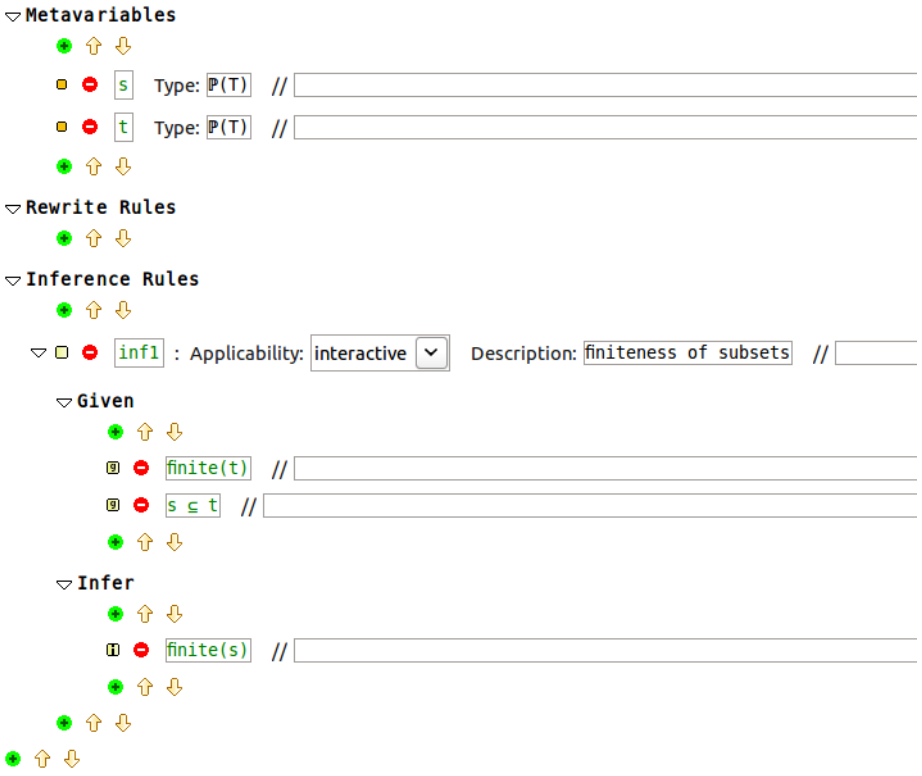


Figure 5.8: An Inference Rule

- *Variables check*: this ensures that the only free variables of the theorem are type parameters.

5. **Rewrite Rule Checks**.
   - *Parsing and type checking of the left hand side*,
   - *Left hand side is not a variable check*,
   - *Presence of at least one right hand side*,
   - *Variables check*: this ensures that the right hand side only refers to variables occurring in the left hand side,
   - *Syntactic class check*: this ensures that right hand sides are of the same syntactic class of the left hand side.
   - *Sides type check*: this ensures that both side of the rule have the same Event-B type.
6. **Inference Rule Checks**.
   - *Presence of infer clause*: each inference rule must have an infer which is syntactically different from $\bot$.
   - *Parsing and type checking of clauses*,
   - *Applicability check*: to ensure that the inference rule is applicable in at least one direction.

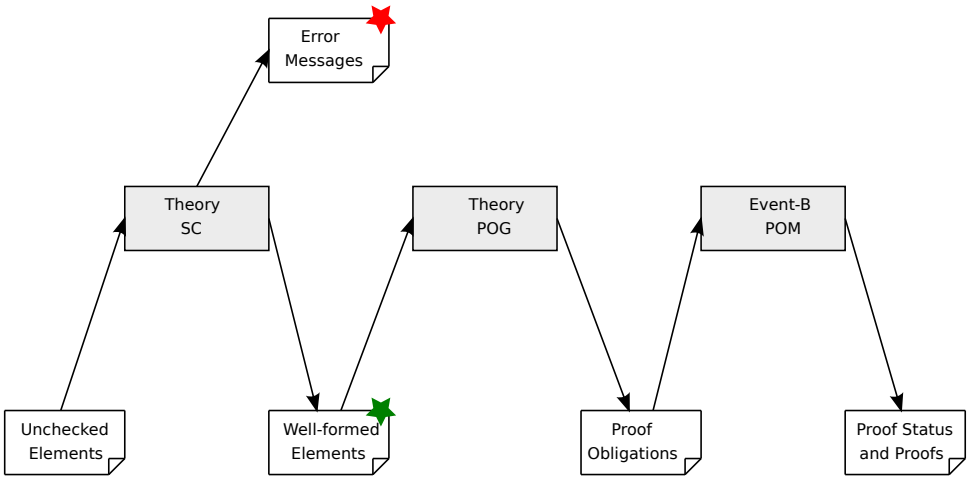


Figure 5.9: Tool-Chain for Event-B Theories

5.1.3 Theory Proof Obligation Generation

Event-B theories are subject to proof obligation generation. The theory proof obligation generator inspects the statically checked theory files (with file extension '.tcf'), and generates the appropriate proof obligation for each inspected element. The generated proof obligations are described in Chapter 4. In summary, the tooling provided for theories follows the same approach of Rodin (see Figure 2.5), and is described in Figure 5.9.

5.1.4 Theory Deployment

Theory development is carried out separately from modelling. This is driven by the different natures of modelling and meta-reasoning. In a typical development, theories are created and organised in hierarchies. Ideally, each theory should define one major mathematical structure, e.g., a sequence, and any supporting operators and proof rules. If a cross-structure theory is required, a different theory can be created for such purpose, and the import directive enables such theory to refer to any required theories. Proof obligations generated from theories should be discharged by the user to ensure soundness preservation[1].

Theory deployment is the process by which theories become available for modelling. By 'availability for modelling', we mean that mathematical and proof extensions can be used when developing models and performing proofs related to them. This is a seamless process; no further actions are required from the end-user. Technically speaking, theory deployment creates the deployed theory file (with file extension '.dtf') which is an exact copy of the statically checked theory file[2].

In contrast with static checking and proof obligation generation, theory deployment is a process initiated by the user. Dependencies between theories (by means of the import directive) are

[1] This, however, is not enforced by the tool. At an early stage of the plug-in lifetime, the enforcement of such requirement may have hindered tool flexibility as far as the user is concerned. Future releases of the Theory plug-in may enforce this particular good practise.

[2] The reader may wonder about the need for another file if it is just an exact copy. The motivation behind this design decision is to keep modelling and meta-reasoning as separate activities. The statically checked theory file is used for meta-reasoning, and the deployed theory file is used for modelling.

automatically observed by the deployment process. The deployment process of a theory ensures that its imported theories are also deployed, thus creating a hierarchy of deployed theories that mirrors the statically checked theory hierarchy. Theory deployment achieves the following two objectives:

1. it allows the end-user to inspect theories for soundness issues by observing the status of proof obligations, and
2. it decouples modelling and meta-reasoning. Deployed theories are the only theories available for use in models.

Figure 5.10 shows the deployment wizard in action.

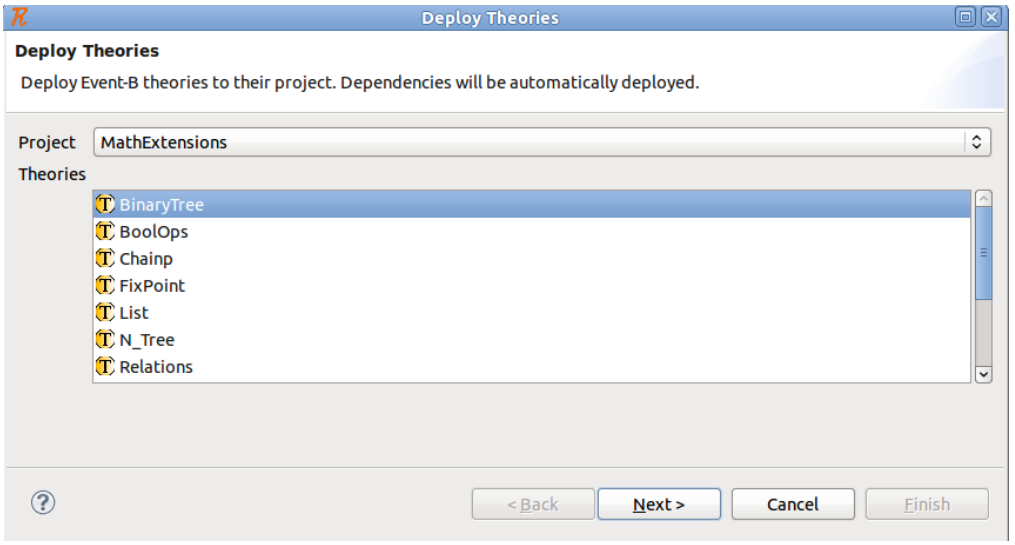


Figure 5.10: The Deployment Wizard

5.1.5 Loading Extensions

Mathematical and proof extensions are loaded from deployed theories. Theories can have one of the following two scopes:

1. **Global Scope**. Also known as 'workspace scope'. This refers to theories which are part of a designated global project[3]. Mathematical and proof extensions in the global theories are available for all projects.
2. **Project Scope**. Also know as 'local scope'. This refers to theories which are part of projects other than the global project. Mathematical and proof extensions in local theories are only available for models in their corresponding project.

Loading extensions is a process initiated by the tool. However, the user can exercise control over what gets loaded by editing/modifying theories. The rational behind scoping theories is the

[3]In the current version of the Theory plug-in (1.3.1), the global project is called 'MathExtensions'. This, however, may change in future releases.

following. Some theories are general enough to be provided as part of a library e.g., sequence, lists and order. These theories should have a global scope. Other theories may be project specific, and as such should have a local scope.

5.1.6 Proof Support

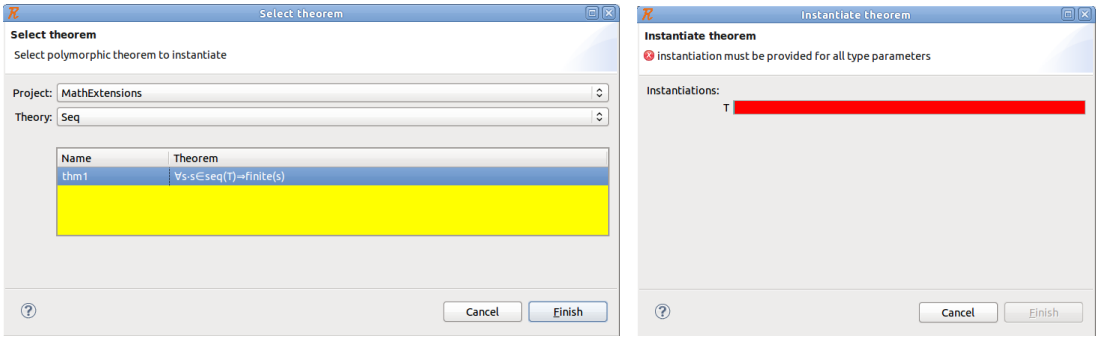
The Theory plug-in provides a mechanism for applying rules and using polymorphic theorems. The Rule-based Prover [78] (known as RbP in the tool) is a contribution to the proof infrastructure of Rodin, and provides a number of reasoners and tactics. An important component of the Rule-based Prover is the pattern matching engine. A particularly interesting aspect of this engine is the associative and associative commutative (AC) matching routine which is inspired by works in [26, 47, 48].[4]

5.1.6.1 Rewriting and Inference

Rewrite and inference rules specified in theories are usable in the same way as existing rewrite and inference rules[5].

5.1.6.2 Polymorphic Theorems

In order to use a polymorphic theorem, an appropriate type instantiation is required. By 'appropriate', we mean that the type instantiation should only refer to types recognised in the sequent to prove (i.e., recognised carrier sets or any of the built-in types $BOOL$ and $\mathbb{Z}$). Figure 5.11 shows the wizard used to select and instantiate a polymorphic theorem.



(a) Select a theorem (b) Instantiate a theorem

Figure 5.11: Using Polymorphic Theorems

The selected and instantiated theorem becomes a visible hypothesis in the current sequent.

[4] A full AC matching procedure is implemented as part of RbP.

[5] This usually is achieved through a hyperlink or a drop-down menu next to the goal or hypothesis predicate.

5.1.6.3 Other Useful Tactics

It is, in some cases, useful to expand the definitions (i.e., rewrite to definition) of all operators used in a sequent. A tactic is provided for this purpose. It attempts to rewrite as much as possible any theory operators with the exception of recursively defined operators and datatype-related expressions (e.g., constructors).

5.2 Summary

In this chapter, we provided an extended overview of the Theory plug-in. We described the theory component which acts as a place holder for the different extensions. Static checking and proof obligation generation are extended to check and validate theories. Deployment makes theories immediately usable in models and proofs. The Theory plug-in implements the ideas presented in this thesis, and it can also be used for other purposes such as code generation [46]. The Theory plug-in was developed as part of the tooling package of the Deploy project [1] which aims to facilitate deployment of formal methods in the industry. The tooling package focused en enhancing the tool support for Event-B by means of Rodin and other useful plug-ins.

Chapter 6

Theory Development: Examples

Chapter 5 provided an extended overview of the Theory plug-in. In this chapter, we provide concrete examples of theories developed using the plug-in. The theories presented in this chapter have been developed to demonstrate the expressiveness of the theory component. Some of the mathematical extensions defined in the forthcoming theories correspond to general mathematics, and are described, in a different way, in the B book [8]. The development of theories presented in this chapter has been a joint effort with Jean-Raymond Abrial.

This chapter is structured in the following way. We present several theories describing useful mathematical structures. We aim to demonstrate by means of examples the different types of extensions (mathematical or proof) that can be expressed. Inductive datatypes and primitive recursive operators are presented for lists.

6.1 Boolean Operators

Expressions and predicates are separate syntactic categories in the Event-B mathematical language. Unlike expressions, predicates do not have a type. However, Event-B provides a boolean type $BOOL$ which has two elements:

$$BOOL = \{TRUE, FALSE\} .$$

$BOOL$, $TRUE$ and $FALSE$ are all expressions. In this section, we introduce a theory *BooleanOps* (Figure 6.1) that defines the different logical connectives $\wedge$, $\vee$ and $\neg$ on boolean types[1]. Note that Event-B also provides an operator *bool* that takes a predicate argument and produces a boolean-typed value according to the truth of the predicate argument.

Theory *BooleanOps* does not introduce type parameters as none is needed to define the required extensions. The theory defines three operators AND, OR and NOT on boolean arguments. The operator definitions are all direct using the *bool* operator. Note that AND and OR are both tagged as associative commutative. This triggers the generation of proof obligations to validate

[1] As opposed to predicates.

theory *BooleanOps*
  operator *AND*
    (**infix**) (**commutative**) (**associative**)
    args $a \in BOOL, b \in BOOL$
    definition $bool(a = TRUE \land b = TRUE)$
  operator *OR*
    (**infix**) (**commutative**) (**associative**)
    args $a \in BOOL, b \in BOOL$
    definition $bool(a = TRUE \lor b = TRUE)$
  operator *NOT*
    (**prefix**)
    args $a \in BOOL$
    definition $bool(a \neq TRUE)$

Figure 6.1: Boolean Operators Theory

the user's claim. Note that operator overloading is not supported in the AST. As such existing syntax symbols (i.e., $\land$, $\lor$ and $\neg$ in this case) cannot be used.

The truth table for the new boolean operators can be defined by means of rewrite rules. We illustrate this for the case of the *NOT* and *AND* operator (Figure 6.2 and Figure 6.3 respectively).

rewrite *NotTruthTable1*
    (automatic) (case complete)
    lhs $NOT\ \ TRUE$
    rhs

$\top$	$FALSE$

rewrite *NotTruthTable2*
    (automatic) (case complete)
    lhs $NOT\ \ FALSE$
    rhs

$\top$	$TRUE$

Figure 6.2: *NOT* Truth Table

Theory *BooleanOps* can be used to create models for electronic circuits.

6.2 Sequences

Sequences are important mathematical structures. The sequence operator is part of classical B modelling repertoire. However, it is not pre-built in the Event-B mathematical language. Theory *Sequences* (Figure 6.4) introduces the sequence operator together with some useful operators, polymorphic theorems and rules.

rewrite *AndTruthTable1*
(automatic) (case complete)
lhs $TRUE \;\; AND \;\; TRUE$
rhs

$\top$	$TRUE$

rewrite *AndTruthTable2*
(automatic) (case complete)
lhs $TRUE \;\; AND \;\; FALSE$
rhs

$\top$	$FALSE$

rewrite *AndTruthTable3*
(automatic) (case complete)
lhs $FALSE \;\; AND \;\; TRUE$
rhs

$\top$	$FALSE$

rewrite *AndTruthTable4*
(automatic) (case complete)
lhs $FALSE \;\; AND \;\; FALSE$
rhs

$\top$	$FALSE$

Figure 6.3: AND Truth Table

theory *Sequences*
type parameters T
operator *seq*
(**prefix**)
args $a \in \mathbb{P}(T)$
definition $\{n, f \cdot f \in 1..n \rightarrow a \mid f\}$

Figure 6.4: Sequences Theory

The sequences theory is parametrised by a single type parameter T. The definition of the operator *seq* includes all the total functions to the argument a from contiguous domains of natural numbers starting from 1. Figure 6.5 introduces useful sequence operators and polymorphic theorems. The sequence head and tail are defined for non-empty sequences. Adding elements to a sequence can be achieved by means of the two operators *seqAppend* and *seqPrepend*. The theorems ensure that the different definitions capture the intuitive understanding of sequences e.g., empty set is a sequence and all sequences are finite.

operator *seq1*
(**prefix**)
args $a \in \mathbb{P}(T)$
definition $seq(a) \backslash \varnothing$
operator *emptySeq*
(**prefix**)
definition $\varnothing : \mathbb{Z} \leftrightarrow T$
operator *isSeqEmpty*
(**prefix**)
args $s \in \mathbb{Z} \leftrightarrow T$
condition $s \in seq(T)$
definition $s = emptySeq$
operator *seqSize*
(**prefix**)
args $s \in \mathbb{Z} \leftrightarrow T$
condition $s \in seq(T)$
definition $card(s)$
operator *seqHead*
(**prefix**)
args $s \in \mathbb{Z} \leftrightarrow T$
condition $s \in seq(T) \land s \neq emptySeq$
definition $s(1)$
operator *seqTail*
(**prefix**)
args $s \in \mathbb{Z} \leftrightarrow T$
condition $s \in seq(T) \land s \neq emptySeq$
definition $\lambda i \cdot i \in 1..(seqSize(s) - 1) \mid s(i + 1)$
operator *seqPrepend*
(**prefix**)
args $s \in \mathbb{Z} \leftrightarrow T, e \in T$
condition $s \in seq(T)$
definition $\{1 \mapsto e\} \cup (\lambda i \cdot i \in 2..(seqSize(s) + 1) \mid s(i - 1))$
operator *seqAppend*
(**prefix**)
args $s \in \mathbb{Z} \leftrightarrow T, e \in T$
condition $s \in seq(T)$
definition $s \cup \{(seqSize(s) + 1) \mapsto e\}$
theorem
$\forall s, a \cdot a \subseteq T \land s \in seq(a) \Rightarrow finite(s)$
$\forall s, a, b \cdot a \subseteq T \land a \subseteq b \land s \in seq(a) \Rightarrow s \in seq(b)$
$\forall s, a \cdot a \subseteq T \land s \in seq(a) \land \neg isSeqEmpty(s) \Rightarrow seqTail(s) \in seq(a)$
$\forall s, a, e \cdot a \subseteq T \land s \in seq(a) \Rightarrow seqPrepend(s, e) \in seq(a \cup \{e\})$
$\forall s, a, e \cdot a \subseteq T \land s \in seq(a) \Rightarrow seqAppend(s, e) \in seq(a \cup \{e\})$

Figure 6.5: Sequences Theory Cont.

The theorems defined in the sequences theory can be turned into inference rules. The following two theorems:

$$\begin{aligned} \forall s, a \cdot a \subseteq T \wedge s \in seq(a) &\Rightarrow finite(s) \\ \forall s, a \cdot a \subseteq T \wedge s \in seq(a) \wedge \neg isSeqEmpty(s) &\Rightarrow seqTail(s) \in seq(a) \end{aligned}$$

can be turned into the two inference rules described in Figure 6.6.

---

metavariables
$s \in \mathbb{Z} \leftrightarrow T,\ a \in \mathbb{P}(T)$
inference *seqIsFinite*
(interactive)
given $s \in seq(a)$
infer $finite(s)$
inference *tailIsSeq*
(interactive)
given $s \in seq(a),\ \neg isSeqEmpty(s)$
infer $seqTail(s) \in seq(a)$

Figure 6.6: Sequence Inference Rules

6.3 Relations

Theory *Relations* (Figure 6.7) defines a number of useful operators in the context of order and equivalence relations. The theory defines the following predicate operators:

- *symmetric*, *asymmetric* and *antisymmetric*,
- $reflexive$ and $irreflexive$,
- *transitive*,
- *partial_order* and *well_order*,
- *equivalence*,
- *linear* and *total_order*.

It is easy to see that the following two theorems hold in theory *Relations*:

$$\begin{aligned} partial\_order(\{a \mapsto b \quad | \quad a \subseteq S \wedge b \subseteq S \wedge a \subseteq b\}) \\ \forall f \cdot f \in S \rightarrow T \quad \Rightarrow \quad equivalence(f; f^{-1})\ . \end{aligned}$$

theory *Relations*
  type parameters S, T
  operator *symmetric*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $r = r^{-1}$
  operator *asymmetric*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $r \cap r^{-1} = \varnothing$
  operator *antisymmetric*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $r \cap r^{-1} \subseteq id$
  operator *reflexive*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $dom(r) \lhd id \subseteq r$
  operator *irreflexive*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $r \cap id = \varnothing$
  operator *transitive*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $r; r \subseteq r$
  operator *partial_order*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $reflexive(r) \land antisymmetric(r) \land transitive(r)$
  operator *well_order*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $partial\_order(r) \land (\forall s \cdot s \neq \varnothing \land s \subseteq dom(r) \Rightarrow (\exists y \cdot y \in s \land s \subseteq r[\{y\}]))$
  operator *equivalence*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $reflexive(r) \land symmetric(r) \land transitive(r)$
  operator *linear*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $S \times S \subseteq r \cup r^{-1}$
  operator *total_order*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $partial\_order(r) \land linear(r)$

Figure 6.7: Relations Theory

6.4 Fixpoint and Closure

In the B book [8], Abrial presents a definition for the fixpoint of a set function (also known as a set transformer). Theory *FixpointClosure* (Figure 6.8) defines two operators: fix and cls. Note that in the following theory, the symbol ; denotes forward composition, whereas the symbol $\circ$ denotes backward composition. The theorems defined in the theory have been shown to be valid

---

theory *FixpointClosure*
  type parameters S
  operator *fix*
    (**prefix**)
    args $f \in \mathbb{P}(S) \rightarrow \mathbb{P}(S)$
    definition $inter(\{s \mid f(s) \subseteq s\})$
  operator *cls*
    (**prefix**)
    args $r \in S \leftrightarrow S$
    definition $fix(\lambda s \cdot s \in \mathbb{P}(S \times S) \mid r \cup (s; r))$
  theorem
    $\forall f, s \cdot f \in \mathbb{P}(S) \rightarrow \mathbb{P}(S) \wedge f(s) \subseteq s \Rightarrow fix(f) \subseteq s$
    $\forall f, v \cdot f \in \mathbb{P}(S) \rightarrow \mathbb{P}(S) \wedge (\forall s \cdot f(s) \subseteq s \Rightarrow v \subseteq s) \Rightarrow v \subseteq fix(f)$
    $\forall f \cdot f \in \mathbb{P}(S) \rightarrow \mathbb{P}(S) \wedge (\forall a, b \cdot a \subseteq b \Rightarrow f(a) \subseteq f(b)) \Rightarrow f(fix(f)) = fix(f)$
    $\forall f \cdot f \in \mathbb{P}(S) \rightarrow \mathbb{P}(S) \Rightarrow (\forall t \cdot t = f(t) \Rightarrow fix(f) \subseteq t)$
    $\forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow cls(r) = r \cup (cls(r); r)$
    $\forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow r \subseteq cls(r)$
    $\forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow cls(r); r \subseteq cls(r)$
    $\forall r, s \cdot r \in \mathbb{P}(S \times S) \wedge s \in \mathbb{P}(S \times S) \wedge r \subseteq s \wedge s; r \subseteq s \Rightarrow cls(r) \subseteq s$
    $\forall r, x \cdot r \in \mathbb{P}(S \times S) \wedge r[x] \subseteq x \Rightarrow cls(r)[x] \subseteq x$
    $\forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow cls(r); cls(r) \subseteq cls(r)$
    $\forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow r; cls(r) \subseteq cls(r)$
    $\forall r \cdot r \in \mathbb{P}(S \times S) \Rightarrow cls(r^{-1}) = (cls(r))^{-1}$

Figure 6.8: Fixpoint and Closure Theory

using the plug-in.

The following theories have also been defined using the Theory plug-in:

1. *Bags*: a theory of bags.
2. *Well_Foundation*: a theory of well-founded sets.
3. *Connectivity*: a theory of strong connectivity.
4. *fchains*: a theory of finite chains.
5. *chainp*: a theory of infinite chains.
6. *BinaryTree*: a theory of inductive binary trees, see Appendix C.
7. *N-Tree*: a theory of inductive n-ary trees.

6.5 Inductive Lists

Theory *Lists* defines the list datatype together with some useful operators. Figure 6.9 shows the definition of the list datatype.

theory *Lists*
  type parameters $S,\ T$
  datatype *List*
    type argument T
    constructors
      nil
      $cons(head : T, tail : List(T))$

Figure 6.9: Inductive Lists Theory

The size of lists can be specified using the following operator:

operator *listSize*
  (**prefix**)
  args $l \in List(T)$
  definition
    case l
      $listSize(nil) \ = \ 0$
      $listSize(cons(x_0, l_0)) \ = \ 1 + listSize(l_0)$

Appending to a list can be defined as follows:

operator *append*
  (**prefix**)
  args $l \in List(T),\ e \in T$
  definition
    case l
      $append(nil, e) \ = \ cons(e, nil)$
      $append(cons(x_0, l_0), e) \ = \ cons(x_0, append(l_0, e))$

Reversing a list can be achieved using the following operator:

operator *rev*
  (**prefix**)
  args $l \in List(T)$
  definition
    case l
      $rev(nil) \ = \ nil$
      $rev(cons(x_0, l_0)) \ = \ append(rev(l_0), x_0)$

Applying a total function to elements of a list to produce another list can be achieved using the following operator:

operator *comp*
(**prefix**)
args $l \in List(T),\ f \in T \leftrightarrow S$
condition $f \in T \rightarrow S$
definition
case l
$comp(nil, f) \ = \ nil : List(S)$
$comp(cons(x_0, l_0), f) \ = \ cons(f(x_0), comp(l_0, f))$

Concatenating two lists can be specified using the following operator:

---

operator *conc*
(**infix**) (**associative**)
args $l_1 \in List(T),\ l_2 \in List(T)$
definition
case l_1
$nil\ conc\ l_2 \ = \ l_2$
$cons(x_0, l_0)\ conc\ l_2 \ = \ cons(x_0, conc(l_0, l_2))$

Flattening a list of lists can be achieved using the following operator:

---

operator *flatten*
(**infix**) (**associative**)
args $l \in List(List(T))$
definition
case l
$flatten(nil) \ = \ nil : List(T)$
$flatten(cons(l_0, ll_0)) \ = \ conc(l_0, flatten(ll_0))$

The following theorems can be discharged from the above primitive recursive definitions:

$$
\begin{aligned}
\forall l, f, x \cdot l \in List(T) \wedge f \in T \rightarrow S \wedge x \in T \quad &\Rightarrow \quad comp(append(l, x), f) = append(comp(l, f), f(x)) \\
\forall l, x \cdot l \in List(T) \wedge x \in T \quad &\Rightarrow \quad rev(append(l, x)) = cons(x, rev(l)) \\
\forall l \cdot l \in List(T) \quad &\Rightarrow \quad rev(rev(l)) = l \\
\forall l1 \cdot l1 \in List(T) \Rightarrow (\forall l2 \cdot l2 \in List(T) \quad &\Rightarrow \quad rev(conc(l1, l2)) = conc(rev(l2), rev(l1))) \\
\forall ll, l \cdot ll \in List(List(T)) \wedge l \in List(T) \quad &\Rightarrow \quad flatten(append(ll, l)) = conc(flatten(ll), l)\ .
\end{aligned}
$$

Proof by induction (in the Theory plug-in) is used to prove the aforementioned theorems.

6.6 A Buffer Example

This example is presented in [46]. A theory of arrays is defined in Figure 6.10. The model describes a simple buffer. The model is initially specified using machine **b0**. Machine **b1** is a refinement of machine **b0**, and uses the theory of arrays in Figure 6.10.

---

theory *Array*
  type parameters T
  operator *array*
    (**prefix**)
    args $s \in \mathbb{P}(T)$
    definition $\{n, f \cdot n \in \mathbb{N} \land f \in 0..(n-1) \rightarrow s \mid f\}$
  operator *arrayN*
    (**prefix**)
    args $n \in \mathbb{Z}, s \in \mathbb{P}(T)$
    condition $n \in \mathbb{N} \land finite(s)$
    definition $\{a \mid a \in array(s) \land card(s) = n\}$
  operator *lookup*
    (**prefix**)
    args $a \in \mathbb{Z} \leftrightarrow T, i \in \mathbb{Z}$
    condition $a \in array(T) \land i \in 0..(card(a)-1)$
    definition $a(i)$
  operator *update*
    (**prefix**)
    args $a \in \mathbb{Z} \leftrightarrow T, i \in \mathbb{Z}, x \in T$
    condition $a \in array(T) \land i \in 0..(card(a)-1)$
    definition $a \lhd\!\!+ \{i \mapsto x\}$
  operator *newArray*
    (**prefix**)
    args $n \in \mathbb{Z}, x \in T$
    condition $n \in \mathbb{N}$
    definition $(0..(n-1)) \times \{x\}$

Figure 6.10: Theory of Arrays [46]

The theory of arrays is used in a data refinement step. In machine **b0**, the variable $abuf$ is defined to be a sequence of integers. The invariants in **b0** state that $abuf$ must be a sequence of a particular length. The variable $abuf$ is initialised to the empty sequence ($\varnothing$).

VARIABLES

`abuf`

INVARIANTS

`inv1 :` $\texttt{abuf} \in \texttt{seq}(\mathbb{Z})$

`inv2 :` $\texttt{seqSize}(\texttt{abuf}) \leq \texttt{maxbuf}$

The refinement in machine **b1** introduces the variable $cbuf$ as a data refinement for the abstract variable $abuf$. The concrete variable is specified using the polymorphic operator $arrayN$.

VARIABLES

cbuf

a

b

INVARIANTS

inv1 : $\texttt{cbuf} \in \texttt{arrayN}(\texttt{maxbuf}, \mathbb{Z})$

...

inv6 : $\forall \texttt{i} \cdot \texttt{i} \in (0 \mathinner{..} \texttt{seqSize}(\texttt{abuf})) \Rightarrow \texttt{prj2}(\texttt{abuf})(\texttt{i}) = \texttt{cbuf}((\texttt{a} + \texttt{i}) \bmod \texttt{maxbuf})$

The concrete variable is initialised using the operator *newArray*.

Initialisation

begin

act1 : $\texttt{cbuf} := \texttt{newArray}(\texttt{maxbuf}, 0)$

act2 : $\texttt{a} := 0$

act3 : $\texttt{b} := 0$

end

The polymorphic operators *lookup* and *update* are used to specify the events **Get** and **Put** in machine **m1**.

Event *Put* $\widehat{=}$

refines *Put*

any

x

where

grd1 : $\texttt{x} \in \mathbb{Z}$

grd2 : $\texttt{b} \geq \texttt{a} \Rightarrow \texttt{b} - \texttt{a} < \texttt{maxbuf}$

then

act1 : $\texttt{b} := (\texttt{b} + 1) \texttt{mod} (\texttt{maxbuf} + 1)$

act2 : $\texttt{cbuf} := \texttt{update}(\texttt{cbuf}, \texttt{b} \bmod \texttt{maxbuf}, \texttt{x})$

end

Event *Get* $\widehat{=}$

refines *Get*

any

y

where

grd1 : $\texttt{a} \neq \texttt{b}$

grd3 : $\texttt{y} \in \mathbb{Z}$

grd2 : $\texttt{y} = \texttt{lookup}(\texttt{cbuf}, \texttt{a})$

then

act1 : $\texttt{a} := (\texttt{a} + 1) \bmod \texttt{maxbuf}$

end

6.7 A Reflection

Prior to our work, axiomatic definitions in contexts were the only possible mechanism by which non-polymorphic functions can be introduced in models. Structure such as sequences, bags and stacks are very useful and common modelling elements, but they are absent from the core syntax of Event-B. Furthermore, from our experience of using the Rodin tool, if a new proof rule is required, a bureaucratic process has to be initiated where resources have to be allocated depending on the urgency of the request.

Despite the lack of quantitative data regarding the usage the Theory plug-in, we argue that the practical contributions of this thesis:

1. complement the Event-B methodology and make it a more rounded formalism,
2. provide an appealing platform to end users because it has facilities for meta-reasoning to complement reasoning and modelling in Event-B,
3. reduce the dependency on the Java programming language and specialised knowledge of Rodin architecture,
4. together with the core Event-B formalism, provide an expressive language that is comparable to higher-order logic as discussed in [13].

Significant effort is required to develop sound theories. Theory hierarchies are a useful structuring mechanism to create operator taxonomies as is the practice in Isabelle/HOL [90]. The effort required to create and validate theories can be decomposed into two large phases:

1. Theory specification phase: new datatypes, operators and proof rules are specified. In this phase, particular attention should be paid to specifying any auxiliary operators that facilitate the use of the main newly introduced structures. In the case of the sequence theory, the *seq* operator is the main structure of the theory, and a number of auxiliary operators, e.g., $emptySeq$, $seqHead$ and $seqTail$, are also defined.
2. Theory validation phase: in this phase, proof obligations are considered and discharged by the user. This phase helps with uncovering errors in the specification of operators and proof rules, in the same way that interactive proof can reveal errors in models. Therefore, theory development is an iterative process.

It is a recurring observation that developing sound theories may take at least the same amount of effort as when developing consistent models. However, the major advantage of using theories is the reusability of definitions thanks to their polymorphic nature. The Theory plug-in provides an obvious upgrade on the process of writing Java code to extend the Event-B language and prover. Finally, the familiarity of our approach to users (reactive development, the use of proof obligations and the use of the existing Rodin user interface for specifying and validating theories) ensures that the Theory plug-in is the tool of choice to extend the Event-B language and proof infrastructure.

6.8 Summary

In this chapter, we presented several theories to illustrate the effectiveness of the Theory plug-in. We demonstrated the use of primitive recursion to define simple operators for inductive lists. Development of theories is an ongoing process, and a sizeable effort is required to create useful and sound libraries that enrich the Event-B mathematical language and proof infrastructure. The Theory plug-in provides a platform to define and validate user-defined libraries.

Chapter 7

Future Work & Conclusion

In this chapter, we bring this thesis to a conclusion by summarising its main contributions. We started this thesis by describing the general setting and context of our work. Chapter 3 presented the theoretical contribution of the thesis in the shape of a study unifying well-definedness and rewriting. Chapter 4 described the approach adopted to enhance the extensibility of Event-B's proof infrastructure, andpresented the technicalities of adding support for user-defined operators and datatypes. Chapter 5 provided an overview of the Theory plug-in which implements the ideas described in this thesis. Finally, Chapter 6 presented several examples of theory development using the Theory plug-in.

This chapter is structured in the following way. We start by summarising the main contributions of this thesis. The contributions are of both practical and theoretical nature. Next, we summarise the key aspects of the Theory plug-in which encapsulates our solutions to the extensibility issues outlined in §1.1. Then, we show the areas in which extensions and additions are feasible. Finally, we present a few concluding remarks.

7.1 Summary of Contributions

As described in §1.3, the scope of this thesis unifies formal methods, logic and software engineering. Our work aims at providing a practically usable mechanism by which the formal methodology Event-B toolset can be soundly extended. More succinctly, this thesis makes the following contributions:

1. It has shown how extensibility and configurability of Rodin can be exploited to add useful feature to the Event-B toolset. The Rodin platform and the Event-B modelling notation was conceived with extensibility and adaptability in mind [12, 60]. We argue that these aspects of the Rodin architecture have helped a great deal in realising the ideas presented in this thesis. The use of a dynamic parser as the backbone for the Event-B abstract syntax tree (AST) enabled the mechanism of adding new operators and datatypes. The ease by which tooling (e.g., the static checking tool) can be specified is largely due to the high configurability of the Rodin platform. Other aspects of the architecture of the

Event-B toolset that enabled our work are described in Chapter 5. This contribution is described in Chapter 4 and 5.

2. It has shown how the existing paradigm used in Event-B developments can be used for *meta-reasoning*. Event-B development is carried out by means of contexts and machines. Proof obligations are generated to verify the consistency of the system with respect to a certain behavioural semantics. Meta-reasoning can be carried out using the theory component to specify language and proof extensions. Proof obligations are then used to ensure extensions are conservative with respect to the logic underpinning the Event-B mathematical language. We argue that the familiarity of our approach can be seen as an important aspect of the usability of our tool. This contribution is described in Chapter 5 and 6.

3. It has shown how the use of proof obligations can be lifted to meta-reasoning about extensions to ensure soundness. For each proof and mathematical extension, certain static checks are performed. Soundness checks are carried out by means of proof obligation generation. The adequacy of the generated proof obligations is justified in this thesis as well as in the work of Schmalz [102]. Note that the meta-reasoning available in Rodin thanks to the Theory plug-in does not equate a provision of a meta-model for Event-B. Such effort is carried out as a shallow embedding of Event-B using Isabelle/HOL [102]. This contribution is described in Chapter 4.

4. It has shown how new polymorphic operators can be defined within the theory component. Predicate (i.e., formula) and expression (i.e., term) operators can be specified as part of the theory component. Operators with direct definitions can be specified and their properties validated by means of proof obligations. The proof obligations related to newly introduced operators are justified in this thesis as well as [102]. This contribution is described in Chapter 4.

5. It provided a characterisation of the interaction between rewriting and deduction in a proof system that accounts for potentially ill-defined terms. This is the theoretical contributions of this thesis. It shows how rewriting and deduction can be interleaved in a sound fashion that takes into consideration well-definedness. The notion of well-definedness preservation for rewrites is introduced, and a simple approach of integrating rewriting and inference within the well-definedness preserving sequent calculus used in Event-B is thoroughly justified. This contribution is described in Chapter 3.

6. It provided a basis for reasoning about proof rules by means of proof obligations. The theory component can be used to specify polymorphic theorems and proof rules. Polymorphic theorems are formulae in Event-B that can be used in proofs provided that a suitable type instantiation is supplied. Proof obligations related to theorems ensure they are valid and well-defined. Proof obligations related to rewrite rules ensure they are valid and well-definedness preserving. Proof obligations related to inference rules ensure they are valid and well-defined. The adequacy of the different proof obligations related to proof extensions are justified in Chapter 3 and Chapter 4.

7. It has shown how to achieve prover extensibility without compromising its soundness. The use of proof obligations is paramount to ensuring soundness is preserved. This is evident from the results of Chapter 4.

Overall, this thesis has contributed a reusable approach to language and prover extensibility of Event-B that maintains the following important requirements:

1. *'Ease of Use'*: the tool support which resulted from this thesis provides an effective and practically usable mechanism to specify and reason about extensions. The adopted approach enables the reuse of definitions, and reduces proof effort across multiple developments.
2. *'Soundness Preservation'*: the use of proof obligations to reason about extensions ensures that the user is aware of any potentially unsound extensions.

7.2 Tool Support

The ideas presented in this thesis provided the basis for a Rodin plug-in that offers facilities to extend the mathematical language and the prover. The Theory plug-in (Chapter 5 and [78]) is an Eclipse-based extension that contributes the following capabilities:

1. It enables the specification of new polymorphic operators (both term and formula operators). It statically checks any such extensions, and automatically generates proof obligations to verify operator properties including: well-definedness strength, associativity and commutativity.
2. It enables the specification of new datatypes. Inductive and enumerated datatypes are supported. Primitive recursive operators can also be defined on any previously defined datatype. The usual checks on datatypes are performed statically and do not require proofs. As such, no proof obligations are generated for datatypes.
3. It provides facilities to specify and validate proof rules and polymorphic theorems. Again, proof obligations ensure soundness of any contributed extensions. Note that rewrite rules generated from operator definitions do not have associated proof obligations.
4. It implements the notion of theory deployment. Once deployed, a theory can readily be used in Event-B models. This ensures that theories are inspected for soundness before they are used in models.
5. It provides a mechanism to manage collections of related theories. The IMPORT directive aims to facilitate the creation of theory hierarchies. Theory hierarchies are discussed in §4.2.
6. It enables an effective meta-reasoning where language and proof extensions are defined within the same component since the two types of extensions are intrinsically linked. Proof extensions serve another important purpose. They facilitate reasoning about new operators and datatypes without detour through their definition.

7.3 Future Work

The following items describe the areas in which further research can be carried out as an extension to our work. The items are prioritised according to their immediate importance.

1. **Creation of a Theory Library**. Established formalisms such as Isabelle have a rich set of libraries ranging from simple set theory to complex continuous mathematics. The creation of a library can provide a standard collection of theories that can be used to enrich the modelling activity. Careful consideration should be given to ensure theories are defined in some well-understood hierarchies to facilitate maintenance.
2. **Validating the Rule-based Prover**. The crucial component of the Rule-based Prover includes the pattern matching engine. A Java-based verification of this particular component can be carried out to increase confidence in the tool. An Event-B specification of certain aspects of the prover, e.g., pattern matching and rule application, is also conceivable. The Rule-based Prover can also be improved by employing some optimisation techniques such as rewrite rule selection by introducing priorities.
3. **Enhancing Support for Datatypes**. Currently, the Theory plug-in only supports enumerated and simple datatype definitions. However, mutually recursive datatype definitions could also be supported in future releases. Furthermore, a fundamental study of datatypes in the logic of Event-B could provide the foundation for further work on the subject.
4. **Support for Axiomatic Definitions**. In some cases, a desirable type cannot be defined using the existing type constructors or datatypes. This is certainly the case for the type of real numbers. The real numbers type $\mathbb{R}$ can be defined as an ordered ring with the addition and multiplication operations. An axiomatic type definition can be used to characterise this particular type.
5. **Support for Binder Definitions**. The mathematical language of Event-B includes several binders, notably $\forall$ and $\exists$. The possibility of adding binders can be explored. The theoretical foundations for such extension are described by Schmalz [102]. However, the existing AST infrastructure does not yet support binder extensions.

7.4 Concluding Remarks

In this thesis, we demonstrated an effective approach to achieve prover and language extensibility in Event-B whilst maintaining the soundness of the formalism. The use of proof obligations when defining extensions ensures that theory developers benefit from the reactive approach underpinning the Rodin philosophy. Possible areas of future work including adding support for mutually recursive datatype definitions and binders have been identified. The tool support can further be improved with respect to performance. The Theory plug-in can provide a strong basis for other potential meta-reasoning activities such as code generation [46].

Appendix A

Chapter 3 Proofs

A.1 Proof of Proposition 3.1

Proposition A.1. *Let t be a Σ-term. If σ is a substitution then*

$$\mathcal{D}(\sigma(t)) \;\Leftrightarrow \bigwedge_{x \in \mathcal{V}ar(t)} \mathcal{D}(\sigma(x)) \;\wedge\; \sigma(\mathcal{D}(t))$$

Proof. We proceed by induction on the structure of the term t.

- *Base Case:* $t = y$ such that $y \in V$. In this case, we have to show the following:

$$\mathcal{D}(\sigma(y)) \Leftrightarrow \bigwedge_{x \in \mathcal{V}ar(y)} \mathcal{D}(\sigma(x)) \;\wedge\; \sigma(\mathcal{D}(y)) \tag{A.1}$$

  Since y is a variable and by expanding the definition of $\mathcal{D}$ (see [15, 82] and §2.4.3), the following holds:

$$\begin{aligned} \mathcal{D}(y) &\;\widehat{=}\; \top \\ \mathcal{V}ar(y) &\;\widehat{=}\; \{y\} \\ \sigma(\top) &\;\widehat{=}\; \top \end{aligned}$$

  Consequently, (A.1) can be rewritten to:

$$\mathcal{D}(\sigma(y)) \Leftrightarrow \mathcal{D}(\sigma(y))$$

  which trivially holds.

- *Inductive Case:* $t = f(s_1, ..., s_n)$ such that $f \in F$ and $s_1, ..$ and s_n are all Σ-terms. In this case, we have to show the following:

$$\mathcal{D}(\sigma(f(s_1, ..., s_n))) \Leftrightarrow \bigwedge_{x \in \mathcal{V}ar(f(s_1,...,s_n))} \mathcal{D}(\sigma(x)) \;\wedge\; \sigma(\mathcal{D}(f(s_1, ..., s_n))) \tag{A.2}$$

with the assumption that for all the terms s_i $(1 \leq i \leq n)$:

$$\mathcal{D}(\sigma(s_i)) \Leftrightarrow \bigwedge_{x \in \mathcal{V}ar(s_i)} \mathcal{D}(\sigma(x)) \ \wedge \ \sigma(\mathcal{D}(s_i))$$

Firstly, we have the following properties:

$$\mathcal{D}(f(s_1, ..., s_n)) \quad \widehat{=} \quad \bigwedge_{i=1}^{n} \mathcal{D}(s_i) \ \wedge \ C^f_{s_1,...,s_n} \qquad \text{[see §2.4.3]} \tag{A.3}$$

$$\mathcal{V}ar(f(s_1, ..., s_n)) \quad \widehat{=} \quad \bigcup_{i=1}^{n} \mathcal{V}ar(s_i) \tag{A.4}$$

$$\sigma(f(s_1, ..., s_n)) \quad \widehat{=} \quad f(\sigma(s_1), ..., \sigma(s_n)) \qquad \text{[see (3.1.2)]} \tag{A.5}$$

Using the previous properties, we get the following:

$$\begin{aligned}
& \mathcal{D}(\sigma(f(s_1, ..., s_n))) \\
\Leftrightarrow \quad & \mathcal{D}(f(\sigma(s_1), ..., \sigma(s_n))) && \text{[definition]} \\
\Leftrightarrow \quad & \bigwedge_{i=1}^{n} \mathcal{D}(\sigma(s_i)) \ \wedge \ C^f_{\sigma(s_1),...,\sigma(s_n)} && \text{[see §2.4.3]} \\
\Leftrightarrow \quad & \bigwedge_{i=1}^{n} \langle \bigwedge_{x \in \mathcal{V}ar(s_i)} \mathcal{D}(\sigma(x)) \ \wedge \ \sigma(\mathcal{D}(s_i)) \rangle \wedge \ C^f_{\sigma(s_1),...,\sigma(s_n)} && \text{[induction hypothesis]} \\
\Leftrightarrow \quad & \bigwedge_{x \in \mathcal{V}ar(f(s_1,...,s_n))} \mathcal{D}(\sigma(x)) \wedge \langle \bigwedge_{i=1}^{n} \sigma(\mathcal{D}(s_i)) \wedge \ C^f_{\sigma(s_1),...,\sigma(s_n)} \rangle && \text{[by (A.4)]} \\
\Leftrightarrow \quad & \bigwedge_{x \in \mathcal{V}ar(f(s_1,...,s_n))} \mathcal{D}(\sigma(x)) \wedge \sigma(\mathcal{D}(f(s_1, ..., s_i))) && \text{[see §2.4.3]}
\end{aligned}$$

Therefore, we have shown that:

$$\mathcal{D}(\sigma(f(s_1, ..., s_n))) \Leftrightarrow \bigwedge_{x \in \mathcal{V}ar(f(s_1,...,s_n))} \mathcal{D}(\sigma(x)) \ \wedge \ \sigma(\mathcal{D}(f(s_1, ..., s_n)))$$

□

A.2 Proof of The Instantiation Theorem

Theorem A.1 (The Instantiation Theorem)**.** *Let $l \xrightarrow{c} r$ be a conditional term rewrite rule, and σ be an idempotent substitution.*

1. *If $l \xrightarrow{c} r$ is valid, then the following sequent is provable:*

$$\sigma(c) \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r) \tag{A.6}$$

2. *If $l \xrightarrow{c} r$ is WD-preserving, then the following sequent is provable:*

$$\sigma(c), \mathcal{D}(\sigma(l)) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\sigma(r)) \tag{A.7}$$

Proof.

1. *Proof of Sequent* (3.3): Since the conditional rewrite rule $l \xrightarrow{c} r$ is valid, the following sequent is provable:

$$c \ \vdash_{\mathcal{D}} \ l = r$$

Furthermore, the following sequent is also provable:

$$\vdash_{\mathcal{D}} \ \forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r$$

where $\vec{x}$ are the free variables of l, since we have the following proof tree:

$$\dfrac{\dfrac{c \ \vdash_{\mathcal{D}} \ l = r}{\vdash_{\mathcal{D}} \ (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r} \Rightarrow goal_{\mathcal{D}}; \wedge hyp_{\mathcal{D}}; mon_{\mathcal{D}}}{\vdash_{\mathcal{D}} \ \forall \vec{x} \cdot [(\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r]} \forall goal_{\mathcal{D}}$$

Observe that the formula

$$\forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r$$

is well-defined[1] which means that the following sequent is also provable:

$$\vdash_{\mathcal{D}} \ \mathcal{D}(\forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r)$$

Using the cut rule, we get the following proof tree:

$$\dfrac{\left\{\begin{array}{l} \sigma(c) \ \vdash_{\mathcal{D}} \ \forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r \\ \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(\forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r) \\ \boxed{\sigma(c), \forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r)} \end{array}\right.}{\sigma(c) \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r)} \ cut_{\mathcal{D}}$$

From the above tree, the following two sequents are provable (as per the discussion above):

$$\begin{array}{lll} \sigma(c) & \vdash_{\mathcal{D}} & \forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r \\ \sigma(c) & \vdash_{\mathcal{D}} & \mathcal{D}(\forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r) \end{array}$$

Note that by Proposition 3.1, we have the following:

$$\dfrac{\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c), \forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r)}{\sigma(c), \forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r)} \ goal_{W\,D}$$

[1] Consider the simple case $\mathcal{D}(\phi) \wedge \phi$, we have

$$\mathcal{D}(\mathcal{D}(\phi) \wedge \phi) \Leftrightarrow ((\mathcal{D}(\mathcal{D}(\phi)) \wedge \mathcal{D}(\phi)) \vee (...) \vee (...))$$

The first disjunct is equivalent to $\top$ by (2.22). Therefore, $\mathcal{D}(\phi) \wedge \phi$ is well-defined.

To prove the remaining (boxed) sequent, we proceed as follows. By applying the rules $\forall hyp_{\mathcal{D}}$, $goal_{WD}$, and $\Rightarrow hyp_{\mathcal{D}}$ on the sequent

$$\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c), \forall \vec{x} \cdot (\mathcal{D}(l) \wedge \mathcal{D}(c) \wedge \mathcal{D}(r) \wedge c) \Rightarrow l = r \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r)$$

we get the following sequents:

$$\begin{array}{rcl}
\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c) & \vdash_{\mathcal{D}} & \bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)) \\
\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c), \sigma(l) = \sigma(r) & \vdash_{\mathcal{D}} & \sigma(l) = \sigma(r) \\
\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c), \mathcal{D}(\sigma(l)) & \vdash_{\mathcal{D}} & \sigma(\mathcal{D}(l)) \\
\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c), \mathcal{D}(\sigma(c)) & \vdash_{\mathcal{D}} & \sigma(\mathcal{D}(c)) \\
\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c), \mathcal{D}(\sigma(r)) & \vdash_{\mathcal{D}} & \sigma(\mathcal{D}(r)) \\
\bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \sigma(c) & \vdash_{\mathcal{D}} & \sigma(c)
\end{array}$$

The first, second and sixth sequent of the previous set are provable using rule $hyp_{\mathcal{D}}$. The third, fourth and fifth sequents can be discharged using Proposition 3.1.

2. *Proof of sequent* (3.4): The following sequent

$$\vdash_{\mathcal{D}} \ \forall \vec{x} \cdot [(\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)] \tag{A.8}$$

is provable ($\vec{x}$ are the free variables of l) is provable if the sequent

$$\mathcal{D}(l), c \ \vdash_{\mathcal{D}} \ \mathcal{D}(r)$$

is provable since we have the following proof tree:

$$\dfrac{\dfrac{\mathcal{D}(l), c \ \vdash_{\mathcal{D}} \ \mathcal{D}(r)}{\vdash_{\mathcal{D}} \ (\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)} \Rightarrow goal_{\mathcal{D}}; \wedge hyp_{\mathcal{D}}; mon_{\mathcal{D}}}{\vdash_{\mathcal{D}} \ \forall \vec{x} \cdot [(\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)]} \ \forall goal_{\mathcal{D}}$$

We observe that the sequent

$$\vdash_{\mathcal{D}} \ \mathcal{D}(\forall \vec{x} \cdot [(\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)]) \tag{A.9}$$

is provable because the formula

$$\forall \vec{x} \cdot [(\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)]$$

is well-defined. Using the cut rule and Proposition 3.1 on the sequent

$$\sigma(c), \mathcal{D}(\sigma(l)) \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(r))$$

we get three sequents to discharge. The following two sequents which are immediately provable (as per the discussion above):

$$\begin{aligned} \vdash_{\mathcal{D}} \quad & \forall \vec{x} \cdot [(\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)] \\ \vdash_{\mathcal{D}} \quad & \mathcal{D}(\forall \vec{x} \cdot [(\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)]) \end{aligned}$$

The third sequent is the following:

$$\begin{array}{l} \sigma(c), \sigma(\mathcal{D}(l)), \bigwedge\limits_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \mathcal{D}(\sigma(l)) \\ \forall \vec{x} \cdot [(\mathcal{D}(c) \wedge \mathcal{D}(l) \wedge c) \Rightarrow \mathcal{D}(r)] \\ \vdash_{\mathcal{D}} \\ \mathcal{D}(\sigma(r)) \end{array}$$

To prove the previous sequent, we proceed as follows. By applying the rules $\forall hyp_{\mathcal{D}}$ and $\Rightarrow hyp_{\mathcal{D}}$ (see [82]) on the previous sequent, we get the following sequents:

$$\sigma(c), \sigma(\mathcal{D}(l)), \bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \mathcal{D}(\sigma(l)) \quad \vdash_{\mathcal{D}} \quad \sigma(\mathcal{D}(c)) \tag{A.10}$$

$$\sigma(c), \sigma(\mathcal{D}(l)), \bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \mathcal{D}(\sigma(l)) \quad \vdash_{\mathcal{D}} \quad \sigma(\mathcal{D}(l)) \tag{A.11}$$

$$\sigma(c), \sigma(\mathcal{D}(l)), \bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \mathcal{D}(\sigma(l)) \quad \vdash_{\mathcal{D}} \quad \sigma(c) \tag{A.12}$$

$$\sigma(c), \sigma(\mathcal{D}(l)), \bigwedge_{x \in \mathcal{V}ar(l)} \mathcal{D}(\sigma(x)), \mathcal{D}(\sigma(l)), \sigma(\mathcal{D}(r)) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\sigma(r)) \tag{A.13}$$

It is easy to see that the first three sequents are provable. Regarding sequent A.13, observe the following:

$$\mathcal{V}ar(r) \subset \mathcal{V}ar(l)$$

since $l \xrightarrow{c} r$ is a rewrite rule. It follows that sequent A.13 is provable if the sequent

$$\bigwedge_{x \in \mathcal{V}ar(r)} \mathcal{D}(\sigma(x)), \sigma(\mathcal{D}(r)) \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(r)) \tag{A.14}$$

is provable which clearly is the case thanks to Proposition 3.1.

□

A.3 Proof of The Term WD-Preserving Rewriting Theorem

Theorem A.2 (Term WD-Preserving Rewriting Theorem)**.** *Let $l \xrightarrow{c} r$ be a conditional term rewrite rule, t be a term, p be a position within t, and σ be an idempotent substitution. If $l \xrightarrow{c} r$*

is valid and WD-preserving, then the following two sequents are provable:

$$\sigma(c) \quad \vdash_{\mathcal{D}} \quad t[\sigma(l)]_p = t[\sigma(r)]_p \tag{A.15}$$

$$\mathcal{D}(t[\sigma(l)]_p), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(t[\sigma(r)]_p) \tag{A.16}$$

Proof.

1. *Proof of sequent* (A.15): We proceed by induction on the structure of the term t.
   (a) **Base Case:** t is a variable, $t = x$. In this case (A.15) becomes

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ x[\sigma(l)]_\epsilon = x[\sigma(r)]_\epsilon$$

   since variables have only one position (ϵ the root position). This simplifies to

   $$\sigma(c) \ \vdash_{\mathcal{D}} \ \sigma(l) = \sigma(r)$$

   which is a provable sequent according to Theorem 3.1.
   (b) **Inductive Case:** t is a function, $t = f(t_1, ..., t_n)$. We distinguish the cases $p = \epsilon$ and $p = iq$ for $1 \leq i \leq n$ and some position q.
      i. Case $p = \epsilon$: this case is similar to the base case.
      ii. Case $p = iq$: we assume the following inductive hypothesis (in this case a provable sequent)

      $$\sigma(c) \ \vdash_{\mathcal{D}} \ t_i[\sigma(l)]_q = t_i[\sigma(r)]_q$$

      and we show that

      $$\sigma(c) \ \vdash_{\mathcal{D}} \ f(t_1, ..., t_i[\sigma(l)]_q, ..., t_n) = f(t_1, ..., t_i[\sigma(r)]_q, ..., t_n)$$

      is a provable sequent where $iq = p$. We proceed as follows:

      $$\dfrac{\dfrac{\sigma(c) \vdash_{\mathcal{D}} t_1 = t_1 \ ... \ \boxed{\sigma(c) \vdash_{\mathcal{D}} t_i[\sigma(l)]_q = t_i[\sigma(r)]_q} \ ... \ \sigma(c) \vdash_{\mathcal{D}} t_n = t_n}{\sigma(c) \ \vdash_{\mathcal{D}} \ t_1 = t_1 \wedge ... \wedge t_i[\sigma(l)]_q = t_i[\sigma(r)]_q \wedge ... \wedge t_n = t_n} \wedge goal_{\mathcal{D}}}{\sigma(c) \ \vdash_{\mathcal{D}} \ f(t_1, ..., t_i[\sigma(l)]_q, ..., t_n) = f(t_1, ..., t_i[\sigma(r)]_q, ..., t_n)}$$

      The boxed sequent is provable since it corresponds to the inductive hypothesis.

   □

2. *Proof of sequent* (A.16): We proceed by induction on the structure of the term t.
   (a) **Base Case:** t is a variable, $t = x$. In this case (A.16) becomes

   $$\mathcal{D}(x[\sigma(l)]_\epsilon), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(x[\sigma(r)]_\epsilon)$$

   since variables only have the root position ϵ. This simplifies to

   $$\mathcal{D}(\sigma(l)), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(\sigma(r))$$

   which is a provable sequent according to Theorem 3.1.

(b) **Inductive Case:** t is a function, $t = f(t_1, ..., t_n)$. We distinguish the cases $p = \epsilon$ and $p = iq$ for $1 \leq i \leq n$ and some position q.

i. Case $p = \epsilon$: this case is similar to the base case.

ii. Case $p = iq$: We assume the following inductive hypothesis (in the shape of a provable sequent)

$$\mathcal{D}(t_i[\sigma(l)]_q), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(t_i[\sigma(r)]_q)$$

and we show that

$$\mathcal{D}(f(t_1, ..., t_i[\sigma(l)]_q, ..., t_n)), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(f(t_1, ..., t_i[\sigma(r)]_q, ..., t_n)) \quad \text{(A.17)}$$

is a provable sequent where $iq = p$. Sequent A.17 can be reduced to the following two sequents:

$$\mathcal{D}(t_i[\sigma(l)]_q), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(t_i[\sigma(r)]_q) \quad \text{(A.18)}$$

$$C^f_{t_1,...,t_i[\sigma(l)]_q,...,t_n}, \sigma(c) \quad \vdash_{\mathcal{D}} \quad C^f_{t_1,...,t_i[\sigma(r)]_q,...,t_n} \quad \text{(A.19)}$$

Sequent A.18 is provable since it is the inductive hypothesis. Sequent A.19 is provable using the first sequent of this theorem, i.e.,

$$\sigma(c) \ \vdash_{\mathcal{D}} \ t[\sigma(l)]_p = t[\sigma(r)]_p$$

□

A.4 Proof of Sequent 3.8

$$\mathcal{D}(f[\sigma(l)]_p), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(f[\sigma(r)]_p)$$

Proof. 1. **Base Case:** f is of the shape $s(t_1, ..., t_n)$ such that $s \in P$ and $t_1, ..., t_n$ are terms. In this case, position p can only be of the form iq for some position q and $1 \leq i \leq n$ since the root position is of a formula. Therefore, (3.8) becomes

$$\mathcal{D}(s(t_1, ..., t_n)[\sigma(l)]_p), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(s(t_1, ..., t_n)[\sigma(r)]_p)$$

where $p = iq$ for some position q and $1 \leq i \leq n$. This can be rewritten as

$$\mathcal{D}(s(t_1, ..., t_i[\sigma(l)]_q, ..., t_n), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(s(t_1, ..., t_i[\sigma(r)]_q, ..., t_n)) \quad \text{(A.20)}$$

Sequent A.20 can be simplified to the following sequent

$$\mathcal{D}(t_i[\sigma(l)]_q), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(t_i[\sigma(r)]_q)$$

whose provability follows immediately from Theorem 3.2.

2. **Inductive Case:** f is of the shape $\varphi \wedge \psi$ such that φ and ψ are formulae. In this case, (3.8) becomes

$$\mathcal{D}((\varphi \wedge \psi)[\sigma(l)]_p), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}((\varphi \wedge \psi)[\sigma(r)]_p) \quad \text{(A.21)}$$

Position p can only be of the form $p = 1q$ or $p = 2q$ for some position q. We distinguish the two cases:

(a) $p = 1q$: In this case, Sequent A.21 becomes

$$\mathcal{D}((\varphi[\sigma(l)]_q \wedge \psi)), \sigma(c) \;\vdash_{\mathcal{D}}\; \mathcal{D}((\varphi[\sigma(r)]_q \wedge \psi)) \tag{A.22}$$

To proceed, we assume that the following sequent is provable:

$$\mathcal{D}((\varphi[\sigma(l)]_q)), \sigma(c) \;\vdash_{\mathcal{D}}\; \mathcal{D}((\varphi[\sigma(r)]_q)) \tag{A.23}$$

and we show that Sequent A.22 is provable. Recall from §2.4.3, we have the following:

$$\mathcal{D}(\varphi \wedge \psi) \mathrel{\widehat{=}} (\mathcal{D}(\varphi) \wedge \mathcal{D}(\psi)) \vee (\mathcal{D}(\varphi) \wedge \neg\varphi) \vee (\mathcal{D}(\psi) \wedge \neg\psi)$$

By applying the previous expansion on Sequent A.22, we obtain the following sequent:

$$\begin{array}{l}(\mathcal{D}(\varphi[\sigma(l)]_q) \wedge \mathcal{D}(\psi)) \vee (\mathcal{D}(\varphi[\sigma(l)]_q) \wedge \neg\varphi[\sigma(l)]_q) \vee (\mathcal{D}(\psi) \wedge \neg\psi), \sigma(c)\\ \vdash_{\mathcal{D}}\\ (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \mathcal{D}(\psi)) \vee (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q) \vee (\mathcal{D}(\psi) \wedge \neg\psi)\end{array}$$

Next, we apply rule $\vee hyp_{\mathcal{D}}$ (i.e., case split), we obtain the following three sequents:

$$\begin{array}{l}\mathcal{D}(\varphi[\sigma(l)]_q), \mathcal{D}(\psi), \sigma(c)\\ \vdash_{\mathcal{D}}\\ (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \mathcal{D}(\psi)) \vee (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q) \vee (\mathcal{D}(\psi) \wedge \neg\psi)\end{array}$$

$$\begin{array}{l}\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c)\\ \vdash_{\mathcal{D}}\\ (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \mathcal{D}(\psi)) \vee (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q) \vee (\mathcal{D}(\psi) \wedge \neg\psi)\end{array}$$

$$\begin{array}{l}\mathcal{D}(\psi), \neg\psi, \sigma(c)\\ \vdash_{\mathcal{D}}\\ (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \mathcal{D}(\psi)) \vee (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q) \vee (\mathcal{D}(\psi) \wedge \neg\psi)\end{array}$$

By applying the analogous rules $\vee goal1_{\mathcal{D}}$ and $\vee goal2_{\mathcal{D}}$ on the previous three sequents, we obtain the following three sequents:

$$\mathcal{D}(\varphi[\sigma(l)]_q), \mathcal{D}(\psi), \sigma(c) \;\vdash_{\mathcal{D}}\; (\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \mathcal{D}(\psi)) \tag{A.24}$$

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \;\vdash_{\mathcal{D}}\; ((\mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q) \tag{A.25}$$

$$\mathcal{D}(\psi), \neg\psi, \sigma(c) \;\vdash_{\mathcal{D}}\; (\mathcal{D}(\psi) \wedge \neg\psi) \tag{A.26}$$

It can easily be seen that Sequent A.26 is provable. We, now, establish the provability of Sequent A.24 and A.25. By applying rule $\wedge goal_{\mathcal{D}}$ on Sequent A.24, we obtain the

following two sequents:

$$\mathcal{D}(\varphi[\sigma(l)]_q), \mathcal{D}(\psi), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\varphi[\sigma(r)]_q) \tag{A.27}$$

$$\mathcal{D}(\varphi[\sigma(l)]_q), \mathcal{D}(\psi), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\psi) \tag{A.28}$$

Sequent A.28 is immediately provable thanks to rule $hyp_{\mathcal{D}}$ (i.e., goal is in the hypotheses). Sequent A.27 provability follows immediately from the inductive hypothesis.

Concerning Sequent A.25, we proceed as follows. By applying rule $\wedge goal_{\mathcal{D}}$, we obtain the following two sequents:

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}\varphi[\sigma(r)]_q \tag{A.29}$$

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \neg\varphi[\sigma(r)]_q \tag{A.30}$$

Sequent A.29 follows from the inductive hypothesis. Sequent A.30 can be shown to be provable thanks to the first part of the theorem (i.e., Sequent 3.7).

(b) $p = 2q$: analogous to the previous case.

3. **Inductive Case:** f is of the shape $\forall x \cdot \varphi$ such that φ is a formula. In this case, (3.8) becomes

$$\mathcal{D}((\forall x \cdot \varphi)[\sigma(l)]_p), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}((\forall x \cdot \varphi)[\sigma(r)]_p) \tag{A.31}$$

In this case, position p can only be of the form $1q$ for some position q since the root position is of a formula. As such, Sequent A.31 can be rewritten to

$$\mathcal{D}(\forall x \cdot \varphi[\sigma(l)]_q), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(\forall x \cdot \varphi[\sigma(r)]_q) \tag{A.32}$$

To proceed, we assume the provability of the following sequent

$$\mathcal{D}(\varphi[\sigma(l)]_q), \sigma(c) \ \vdash_{\mathcal{D}} \ \mathcal{D}(\varphi[\sigma(r)]_q) \tag{A.33}$$

and we show the provability of Sequent A.32. Recall from §2.4.3, we have the following:

$$\mathcal{D}(\forall x \cdot \varphi) \mathrel{\widehat{=}} (\forall x \cdot \mathcal{D}(\varphi)) \vee (\exists x \cdot \mathcal{D}(\varphi) \wedge \neg\varphi)$$

By applying the previous expansion on Sequent A.32, we get the following sequent:

$$\begin{aligned}&(\forall x \cdot \mathcal{D}(\varphi[\sigma(l)]_q)) \vee (\exists x \cdot \mathcal{D}(\varphi[\sigma(l)]_q) \wedge \neg\varphi[\sigma(l)]_q), \sigma(c)\\ &\vdash_{\mathcal{D}}\\ &(\forall x \cdot \mathcal{D}(\varphi[\sigma(r)]_q)) \vee (\exists x \cdot \mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q)\end{aligned}$$

By applying rule $\vee hyp_{\mathcal{D}}$ (i.e., case split) on the previous sequent, we obtain the following two sequents:

$$\begin{array}{l}\forall x \cdot \mathcal{D}(\varphi[\sigma(l)]_q), \sigma(c)\\ \vdash_{\mathcal{D}}\\ (\forall x \cdot \mathcal{D}(\varphi[\sigma(r)]_q)) \vee (\exists x \cdot \mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q)\end{array}$$

$$\begin{array}{l}\exists x \cdot (\mathcal{D}(\varphi[\sigma(l)]_q) \wedge \neg\varphi[\sigma(l)]_q), \sigma(c)\\ \vdash_{\mathcal{D}}\\ (\forall x \cdot \mathcal{D}(\varphi[\sigma(r)]_q)) \vee (\exists x \cdot \mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q)\end{array}$$

By applying the analogous rules $\vee goal1_{\mathcal{D}}$ and $\vee goal2_{\mathcal{D}}$ on the previous two sequents, we obtain the following two sequents:

$$\forall x \cdot \mathcal{D}(\varphi[\sigma(l)]_q), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \forall x \cdot \mathcal{D}(\varphi[\sigma(r)]_q) \tag{A.34}$$

$$\exists x \cdot \mathcal{D}(\varphi[\sigma(l)]_q) \wedge \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \exists x \cdot \mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q \tag{A.35}$$

Firstly, we show the provability of Sequent A.34. By applying rule $\forall goal_{\mathcal{D}}$ (note that the side condition holds thanks to the proviso of the theorem), we obtain the following

$$\forall x \cdot \mathcal{D}(\varphi[\sigma(l)]_q), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\varphi[\sigma(r)]_q)$$

Next, by applying the rule $\forall hyp_{\mathcal{D}}$ on the previous sequent, we get the following two sequents:

$$\forall x \cdot \mathcal{D}(\varphi[\sigma(l)]_q), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(x) \tag{A.36}$$

$$\mathcal{D}(\varphi[\sigma(l)]_q), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\varphi[\sigma(r)]_q) \tag{A.37}$$

The provability of Sequent A.36 follows from the fact that variables are well-defined. The provability of Sequent A.37 follows from the inductive hypothesis.

Secondly, we show the provability of Sequent A.35. By applying the rule $\exists hyp_{\mathcal{D}}$ (note that the side condition holds thanks to the proviso of the theorem), we obtain the following

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \exists x \cdot \mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q \tag{A.38}$$

Next, by applying rule $\exists goal_{\mathcal{D}}$ on Sequent A.38, we obtain the following two sequents

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(x) \tag{A.39}$$

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\varphi[\sigma(r)]_q) \wedge \neg\varphi[\sigma(r)]_q \tag{A.40}$$

The provability of Sequent A.39 follows from the fact that variables are well-defined. We conclude this proof by showing the provability of Sequent A.40. We proceed as follows. By applying rule $\wedge goal_{\mathcal{D}}$, we obtain the following two sequents

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\varphi[\sigma(r)]_q) \tag{A.41}$$

$$\mathcal{D}(\varphi[\sigma(l)]_q), \neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \neg\varphi[\sigma(r)]_q \tag{A.42}$$

which can, respectively, be simplified to

$$\mathcal{D}(\varphi[\sigma(l)]_q), \sigma(c) \quad \vdash_{\mathcal{D}} \quad \mathcal{D}(\varphi[\sigma(r)]_q) \tag{A.43}$$

$$\neg\varphi[\sigma(l)]_q, \sigma(c) \quad \vdash_{\mathcal{D}} \quad \neg\varphi[\sigma(r)]_q \tag{A.44}$$

Sequent A.43 is provable since it corresponds to the inductive hypothesis. Sequent A.44 is provable thanks to the first part of this theorem (i.e., Sequent 3.7).

□

Appendix B

Buffer Case Study

This appendix provides listing of contexts and machines in the case study appearing in Chapter 6. The following listing describes the context used in the model.

CONTEXT c0
CONSTANTS
maxbuf
AXIOMS
axm1 : maxbuf $\in \mathbb{N}$
axm2 : maxbuf $= 20$
END

The following listing describes the first abstraction of the buffer using sequences.

MACHINE b0
SEES c0
VARIABLES
abuf
INVARIANTS
inv1 : abuf $\in$ seq($\mathbb{Z}$)
inv2 : seqSize(abuf) $\leq$ maxbuf
EVENTS
Initialisation
extended
begin
act1 : abuf $:=$ empty
end
Event *Put* $\widehat{=}$
any
x

where

grd1 : $\mathtt{x} \in \mathbb{Z}$

grd2 : $\mathtt{seqSize}(\mathtt{abuf}) < \mathtt{maxbuf}$

then

act1 : $\mathtt{abuf} := \mathtt{seqAppend}(\mathtt{abuf}, \mathtt{x})$

end

Event *Get* $\widehat{=}$

any

y

where

grd1 : $\neg\mathtt{seqIsEmpty}(\mathtt{abuf})$

grd2 : $\mathtt{y} = \mathtt{seqHead}(\mathtt{abuf})$

then

act1 : $\mathtt{abuf} := \mathtt{seqTail}(\mathtt{abuf})$

end

END

The following listing describes the first refinement of the machine **b0** using arrays. Note that the operator *mod* refers to the arithmetic modulo operator.

MACHINE b1

REFINES b0

SEES c0

VARIABLES

cbuf

a

b

INVARIANTS

inv1 : $\mathtt{cbuf} \in \mathtt{arrayN}(\mathtt{maxbuf}, \mathbb{Z})$

inv2 : $\mathtt{a} \in \mathbb{Z}$

inv3 : $\mathtt{b} \in \mathbb{Z}$

inv4 : $\mathtt{a} \in 0 \mathinner{..} \mathtt{maxbuf} - 1$

inv5 : $\mathtt{b} \in 0 \mathinner{..} \mathtt{maxbuf}$

inv6 : $\forall \mathtt{i} \cdot \mathtt{i} \in (0 \mathinner{..} \mathtt{seqSize}(\mathtt{abuf})) \Rightarrow \mathtt{prj2}(\mathtt{abuf})(\mathtt{i}) = \mathtt{cbuf}((\mathtt{a} + \mathtt{i}) \bmod \mathtt{maxbuf})$

EVENTS

Initialisation

begin

act1 : $\mathtt{cbuf} := \mathtt{newArray}(\mathtt{maxbuf}, 0)$

act2 : $\mathtt{a} := 0$

act3 : $\mathtt{b} := 0$

end

Event *Put* $\widehat{=}$

refines *Put*

any
x
where
grd1 : $x \in \mathbb{Z}$
grd2 : $b \geq a \Rightarrow b - a < maxbuf$
then
act1 : $b := (b + 1) mod (maxbuf + 1)$
act2 : $cbuf := update(cbuf, b \bmod maxbuf, x)$
end

Event *Get* $\widehat{=}$
refines *Get*
any
y
where
grd1 : $a \neq b$
grd3 : $y \in \mathbb{Z}$
grd2 : $y = lookup(cbuf, a)$
then
act1 : $a := (a + 1) \bmod maxbuf$
end

END

Appendix C

Binary Trees Theory

This appendix lists a simple theory of binary trees that was developed in collaboration with Jean-Raymond Abrial.

theory *BinaryTree*
  type parameters T
  datatype *Tree*
    type argument T
    constructors
      empty
      $tree(left : Tree(T), val : T, right : Tree(T))$
  operator *treeDepth*
    (**prefix**)
    args $t \in Tree(T)$
    definition
      case l
        $treeDepth(empty) \;=\; 0$
        $treeDepth(tree(l, x, r)) \;=\; 1 + max\{treeDepth(l), treeDepth(r)\}$
  operator *mirror*
    (**prefix**)
    args $t \in Tree(T)$
    definition
      case l
        $mirror(empty) \;=\; empty$
        $mirror(tree(l, x, r)) \;=\; tree(mirror(r), x, mirror(l))$
  theorem
    $\forall t.t \in Tree(T) \Rightarrow mirror(mirror(t)) = t$

References

[1] Deploy: Industrial deployment of system engineering methods providing high dependability and productivity, February 2008. http://www.deploy-project.eu/html/about_deploy_project.html.

[2] VDMTools: advances in support for formal modeling in VDM. *SIGPLAN Not.*, 43:3–11, February 2008.

[3] Rodin 2.0 Release Notes. Systerel, France, 2010. http://wiki.event-b.org/index.php/Rodin_Platform_2.0_Release_Notes.

[4] Camille Editor, 2011. http://wiki.event-b.org/index.php/Camille_Editor.

[5] SMT Solvers Plug-in. Systerel, France, 2011. http://wiki.event-b.org/index.php/SMT_Solvers_Plug-in.

[6] Atelier B, the industrial tool to efficiently deploy the B Method. Clearsy, France, 2012. http://www.atelierb.eu/en/.

[7] Jean R. Abrial and Dominique Cansell. Click'n Prove: Interactive Proofs within Set Theory. In *Lecture Notes in Computer Science : Theorem Proving in Higher Order Logics*, pages 1–24, 2003.

[8] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[9] Jean-Raymond Abrial. A System Development Process with Event-B and the Rodin Platform. In Michael Butler, Michael Hinchey, and Mara Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, volume 4789 of *Lecture Notes in Computer Science*, pages 1–3. Springer Berlin / Heidelberg, 2007.

[10] Jean-Raymond Abrial. Formal Methods: Theory Becoming Practice. *Journal of Universal Computer Science*, 13(5):619–628, May 2007.

[11] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering (1st ed.)*. Cambridge University Press, New York, NY, USA, 2010.

[12] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In *ICFEM 2006, LNCS*, pages 588–605. Springer, 2006.

[13] Jean-Raymond Abrial, Dominique Cansell, and Guy Laffitte. "Higher-Order" Mathematics in B. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 370–393. Springer-Verlag, 2002.

[14] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.*, 77:1–28, January 2007.

[15] Jean-Raymond Abrial and Louis Mussat. On Using Conditional Definitions in Formal Theories. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 242–269. Springer-Verlag, 2002.

[16] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. Efficient Well-Definedness Checking. In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 100–115. Springer-Verlag, 2008.

[17] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[18] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification, 2000.

[19] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[20] Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *Proceedings of the 4th international conference on Formal Specification and Development in Z and B*, ZB'05, pages 334–354. Springer-Verlag, 2005.

[21] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering, Number 1783 In LNCS*, pages 363–366. Springer, 2000.

[22] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, pages 364–387, 2005.

[23] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: an overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69. Springer-Verlag, 2005.

[24] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984. 10.1007/BF00264250.

[25] Patrick Behm, Lilian Burdy, and Jean-Marc Meynadier. Well Defined B. In *B '98: Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 29–45. Springer-Verlag, 1998.

[26] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *J. Symb. Comput.*, 3(1-2):203–216, February 1987.

[27] Stefan Berghofer and Markus Wenzel. Inductive Datatypes in HOL - Lessons Learned in Formal-Logic Engineering. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '99, pages 19–36. Springer-Verlag, 1999.

[28] Juan C. Bicarregui, John S. Fitzgerald, Peter A. Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: a practitioner's guide*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.

[29] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, apr 1995.

[30] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005.

[31] Michael Butler and Stefan Hallerstede. The Rodin Formal Modelling Tool. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, December 2007.

[32] J. H. Cheng. *A Logic for Partial Functions*. PhD Thesis, University of Manchester, 1986.

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

[34] Alonzo Church. A Formulation of the Simple Theory of Types. *J. Symb. Log.*, 5(2):56–68, 1940.

[35] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 2001.

[36] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[37] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42. Springer-Verlag, 2009.

[38] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[39] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: industrial usage. *Software Engineering, IEEE Transactions on*, 21(2):90 –98, Feb 1995.

[40] Istituto Trentino Di Cultura, Alessandro Armando, Alessandro Armando, Alessandro Cimatti, and Alessandro Cimatti. Building and executing proof strategies in a formal metatheory. In *Advances in Artifical Intelligence: Proceedings of the Third Congress of the Italian Association for Artificial Intelligence, IA*AI'93, Volume 728 of Lecture Notes in Computer Science*, pages 11–22. Springer-Verlag, 1993.

[41] Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3:69–115, February 1987.

[42] Nachum Dershowitz. Term Rewriting Systems by (Marc Bezem, Jan Willem Klop, and Roel de Vrijer, eds.), Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 55, 2003, hard cover: ISBN 0-521-39115-6, xxii+884 pages. *Theory and Practice of Logic Programming*, 5:395–399, 2005.

[43] Nachum Dershowitz and David A. Plaisted. Rewriting. In *Handbook of Automated Reasoning*, pages 535–610. 2001.

[44] David L. Detlefs. An overview of the extended static checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, 1995.

[45] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning 31(1)*, pages 33–72, 2003.

[46] Andrew Edmunds, Michael Butler, Issam Maamria, Renato Silva, and Chris Lovell. Event-B Code Generation: Type Extension with Theories. In *ABZ'2012*, 2012 (In Press).

[47] Steven Eker. Associative-Commutative Matching Via Bipartite Graph Matching. *Comput. J. 38(5)*, pages 381–399, 1995.

[48] Steven Eker. Associative-Commutative Rewriting on Large Terms. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, Lecture Notes in Computer Science, pages 14–29. Springer, June 2003.

[49] Eclipse Foundation. Eclipse Platform. `http://www.eclipse.org/`, 2011.

[50] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row Publishers, Inc., New York, NY, USA, 1985.

[51] Antony Galton. *Temporal logic and computer science: an overview*, pages 1–52. Academic Press Professional, Inc., San Diego, CA, USA, 1987.

[52] François Garillot and Benjamin Werner. Simple Types in Type Theory: Deep and Shallow Encodings. In *TPHOLs*, pages 368–382, 2007.

[53] Martin Giese. Taclets and the KeY Prover. *Electron. Notes Theor. Comput. Sci.*, 103:67–79, November 2004.

[54] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action.* Kluwer, 2000.

[55] Joseph A. Goguen, Claude Kirchner, Hlne Kirchner, Aristide Mgrelis, Jos Meseguer, and Timothy C. Winkler. An Introduction to OBJ 3. In Stphane Kaplan and Jean-Pierre Jouannaud, editors, *Conditional Term Rewriting Systems, 1st International Workshop, Orsay, France, July 8-10, 1987, Proceedings*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 1987.

[56] Robert Goldblatt. Mathematical Modal Logic: A View of Its Evolution. *Journal of Applied Logic*, 1(5-6):309–392, 2003.

[57] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[58] Mike Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic, 1985.

[59] David Griffioen and Marieke Huisman. A comparison of PVS and Isabelle/HOL. In *Theorem Proving in Higher Order Logics, number 1479 in Lect. Notes Comp. Sci*, pages 123–142. Springer, 1998.

[60] Stefan Hallerstede. Justifications for the Event-B Modelling Notation. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 49–63. Springer Berlin / Heidelberg, 2006.

[61] Stefan Hallerstede. On the Purpose of Event-B Proof Obligations. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 125–138. Springer-Verlag, 2008.

[62] John Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz`.

[63] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[64] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, January 1983.

[65] Kryštof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In *Proceedings of the 32nd annual German conference on Advances in artificial intelligence*, KI'09, pages 435–443. Springer-Verlag, 2009.

[66] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. Supporting Reuse in Event B Development: Modularisation Approach. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Rgine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, volume 5977 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2010.

[67] C. B. Jones. Reasoning About Partial Functions in the Formal Development of Programs. *Electron. Notes Theor. Comput. Sci.*, 145:3–25, January 2006.

[68] C. B. Jones and C. A. Middelburg. A Typed Logic of Partial Functions Reconstructed Classically. *ACTA INFORMATICA*, 31:399–430, 1994.

[69] Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall, Upper Saddle River, NJ, USA, 1980.

[70] Cliff B. Jones. *Systematic software development using VDM (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[71] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[72] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative integrating tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35:1–6, January 2010.

[73] P. A. Lindsay, C. B. Jones, K. D. Jones, and R. D. Moore. *Mural: A Formal Development Support System.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.

[74] X. Liu, Z. Chen, H. Yang, H. Zedan, and William C. Chu. A Design Framework for System Re-engineering. In *Proceedings of Asia Pacific and International Computer Science Conference*, pages 324–352, 1997.

[75] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types.* Wiley, 1996.

[76] Gerald Luettgen, Cesar Munoz, Ricky Butler, Ben D Vito, and P. Miner. Towards a customizable PVS. Technical report, 2000.

[77] Issam Maamria and Michael Butler. Rewriting and Well-Definedness within a Proof System. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *PAR*, volume 43 of *EPTCS*, pages 49–64, 2010.

[78] Issam Maamria, Michael Butler, Andrew Edmunds, and Abdolbaghi Rezazadeh. On an Extensible Rule-Based Prover for Event-B. In Marc Frappier, Uwe Glsser, Sarfraz Khurshid, Rgine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 407–407. Springer Berlin / Heidelberg, 2010.

[79] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[80] Farhad Mehta. Supporting Proof in a Reactive Development Environment. *International Conference on Software Engineering and Formal Methods*, 0:103–112, 2007.

[81] Farhad Mehta. A Practical Approach to Partiality - A Proof Based Approach. In *ICFEM*, pages 238–257, 2008.

[82] Farhad Mehta. *Proofs for the Working Engineer.* PhD Thesis, ETH Zurich, 2008.

[83] Christophe Métayer and Laurent Voisin. *The Event-B Mathematical Language (Version 2)*, March 2009.

[84] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[85] Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford, CA, USA, 1972.

[86] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[87] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[88] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[89] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle Logics: HOL, 2000.

[90] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[91] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[92] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-calvert. PVS Language Reference, 2001.

[93] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 772–773. Springer-Verlag, 1988.

[94] Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5:363–397, September 1989.

[95] Lawrence C. Paulson, Tobias Nipkow, and Markus Wenzel. The Isabelle Reference Manual, 2007.

[96] David A. Plaisted. *Equational Reasoning and Term Rewriting Systems*. Oxford University Press, Inc., New York, NY, USA, 1993.

[97] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[98] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12:23–41, January 1965.

[99] Ken Robinson. Reconciling Axiomatic and Model-Based Specifications Reprised. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 223–236. Springer, 2008.

[100] Donald Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Workshops in Computing, pages 99–130. Springer, 1991.

[101] Matthias Schmalz. Export to Isabelle. ETH Zurich, Switzerland, 2009. `http://wiki.event-b.org/index.php/Export_to_Isabelle`.

[102] Matthias Schmalz. The Logic of Event-B. Technical Report 698, ETH Zurich, Switzerland, 2010. `http://www.inf.ethz.ch/research/disstechreps/techreports`.

[103] Matthias Schmalz. Term rewriting in logics of partial functions. In *Proceedings of the 13th international conference on Formal methods and software engineering*, ICFEM'11, pages 633–650. Springer-Verlag, 2011.

[104] Colin Snook. Event-B Records Extension. University of Southampton, UK, 2009. `http://wiki.event-b.org/index.php/Records_Extension`.

[105] Patrick Suppes. *Introduction to Logic*. Dover, 1999.

[106] Sander D. Vermolen, Jozef Hooman, and Peter Gorm Larsen. Proving consistency of VDM models using HOL. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2503–2510. ACM, 2010.

[107] N. Volker. On the Representation of Datatypes in Isabelle/HOL. Technical report, First Isabelle Users Workshop, 1995.

[108] Markus Wenzel. Type Classes and Overloading in Higher-Order Logic. In *TPHOLs*, pages 307–322, 1997.

[109] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[110] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.