Automating Verification of Event-B Models

Paulius Stankaitis

November 20, 2016

Automating Verification of Event-B Models

Paulius Stankaitis

In Partial Fulfillment of the Requirements for the Degree of

Philosophy of Master

1. Examiner Elena Troubitsyna
Turku Centre for Computer Science
Åbo Akademi University

2. Examiner Yamine Ait Ameur
Department of Computing and Applied Mathematics
INP-ENSEEIHT

Supervisors Alexander Romanovsky and Alexei Iliasov

November 20, 2016

Paulius Stankaitis
Automating Verification of Event-B Models
Examiners: Elena Troubitsyna and Yamine Ait Ameur
Supervisors: Alexander Romanovsky and Alexei Iliasov

Newcastle University
Center for Software Reliability
School of Computing Science
Claremont road
Newcastle upon Tyne
NE1 7RU

Abstract

Over the last decades, various techniques have been developed and applied in order to ensure systems correctness. Formal methods are widely regarded as the quintessential approach to specification, development and verification of systems. Though there has been considerable progress that has led to its acceptance in industry, the formal methods community must still overcome several critical obstacles.

A lot of effort in a system validation goes into the formal verification process. However, a significant part of provable conjectures requires proof hints from a user. In particular for larger models, this becomes extremely tedious as identical or similar proofs have to be repeated over and over again, especially after model refactoring stages. Moreover, in the industrial setting, domain engineers are very rarely capable of carrying out a mathematical proof and their training can be very expensive.

The research presented in this thesis attempts to address the problem of interactive proofs. First of all, we try to reduce the number of interactive proof conditions needed to formally verify a model. There are more than a dozen theorem provers which are different on various aspects. To fully exploit the capacity of these automated theorem provers, a common interface is necessary. Theorem proving is a computationally intensive exercise, so, we want to explore the cheap computational power, cloud technology offers, for doing proofs remotely.

In this thesis, we propose a verification tool which supports verification of Event-B proof obligations and is integrated in the Rodin platform. Moreover, we present the architecture and implementation of the proposed verification tool. Furthermore, we also conduct a case study where we evaluate our technique on several Event-B models and further discuss how this approach can be applied in various settings and domains.

Throughout our research we discovered that with the new verification tool we could address another interactive proof problem - fragility and non reusability. There is a number of circumstances when existing interactive proofs become invalidated and a new version of an undischarged proof obligation appears. The solution is to allow a modeller to construct a generic lemma which would be strong enough to discharge a proof obligation, yet, be general enough to be reused in a different context. We

hypothesise that at a certain stage accumulated generic lemmata would dramatically increase automation in the verification process.

Acknowledgement

Throughout this research, I received a great support from many people who directly or indirectly helped me in this interesting and challenging time. First of all, I would like to thank my supervisor, Alexander Romanovsky, for the amazing opportunity to learn and explore this interesting field - formal methods, a continuous support and kind guidance throughout this work.

I am massively grateful to Alexei Iliasov and David Adjepon-Yamoah for their patience, energy and significant contribution in this enjoyable collaboration. I learned so much from them and I am really looking forward to continuing working together.

Also, I would like to thank Loredana Tec, Karolis Kazlaukis and Timothy Hardman for reading the early draft of the thesis and a useful feedback. A special thanks goes to the RSSB SafeCap+ project and to our colleagues from Siemens Rail Automation for invaluable feedback and support.

Lastly, I would like to thank and dedicate this thesis to my amazing family without their encouragement and support I would not have done this.

List of Figures

List of Tables

Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

Newcastle upon Tyne, November 20, 2016

Paulius Stankaitis

Contents

1 Introduction

This chapter gives an overview of the rationale of this research. In particular, we emphasise the importance of formal methods for developing safety-critical systems. It also discusses the formal proof process as a rigorous method to provide evidence about correctness of the system. Moreover, we present challenges of the proof process, in particular interactive proofs in an industrial setting.

In the following sections we outline the thesis structure, contributions and motivation. We establish research objectives and discuss the direction of the preliminary work. A part of this research work was published and presented at various conferences and workshops, hence, in the last section we provide a list of published work.

1.1 Research Motivation

Society relies on engineers who design and validate systems. Throughout recent decades we have been computerising our systems making them more and more complex. And therefore, the tools and methods we used to validate a system in the past are becoming insufficient and not adequate for the present, and future problems.

A subset of systems we classify as safety-critical requires rigorous analysis and validation. There are more than a dozen ways one can define a safety-critical system. Although, perhaps the most fundamental definition is a system, whose failure is likely to have tragic implications on human lives, property and environment. For that reason such systems have to meet the highest requirements in order to be certified and deployed. Safety-critical systems exist in various domains, for example: railway, medical, military and so forth. Therefore, it is within our interest to have generic methods and tools which can ensure the correctness of such systems.

An extensive system testing is one of the most popular methods to provide evidence of systems faultlessness. In particular, for software related projects a moderate cost of testing is extremely appealing to industry [Ili08]. Although, the main drawback of system testing methods is the inability to cover the complete state-space. The paper by John C. Knight on challenges and directions of safety-critical systems concluded that a system validation by testing is an inadequate technique [Kni02].

As he further discussed, alternative techniques like formal verification is highly advisable, however, the current state of such methods have limited applicability.

Formal verification is closely linked with formal specification. A common practice in designing a software system is to produce an executable program in early stages of development. The software implementation is then further refined by thorough testing. Introducing formal methods in the development phase requires a more rigorous thinking about the specifications and requirements before any implementation has been started. One of the first steps in formally designing systems is the development of a mathematical model. The main advantage of this technique is that system specifications and requirements can no longer be ambiguous. Furthermore, developing a mathematical model at the earlier stages of system design can signal potential faults.

One might reason that the formal system development approach may be too costly and time consuming, especially in times when the time-to-market criteria is vital for the businesses. However, several industrial cases have shown that large costs have been saved by spending more time in formally specifying systems [Hal90]. This can be attributed to the fact that better specifications reduced the testing, debugging and redevelopment effort. Furthermore, it has been reported that the formal training costs are one-time expenses and a good future investment.

A lot of effort formally validating systems goes into the verification process. Mathematical proof is vital to confirm that systems properties hold. Furthermore, an exercise of mathematical proof can not only provide more assurance of the system, but also further increase understanding of the system. Even though, a proof can be done manually it can be error-prone. A much preferred option is to mechanise the verification exercise [BS93].

The automated reasoning area has had some great success stories and made a great progress in automating the verification process. However, in the industrial setting, domain engineers are very rarely capable of carrying out a mathematical proof and their training could potentially be expensive. Hence, it is vitally important to automate verification task as much as possible in order to keep formal methods attractive for industry.

Despite of all the progress achieved in the area of automated theorem proving, interactive proofs still remain a big challenge. Especially for large industrial cases where even a small percentage of total interactive proofs can results in months of man effort. A widely used example of a safety-critical system successfully developed using B formal method is the Paris Metro Line 14 [Abr07]. Merely a fraction of all proofs were interactive proof in that project, however, it still took approximately

seven months work to complete. Further analysis and overview of several research, and industrial projects where formal methods were used are discussed by Velykis [Vel15]. The complexity and amount of interactive proofs remain one of the biggest verification challenges and a "push-button" verification solution is desired.

As hardware prices continue to drop, the cloud paradigm has emerged as an alternative to do computations remotely. The cheap computational resources made available by data centres and cloud computing opens possibilities to complete CPU and memory intensive tasks remotely and in parallel.

To summarise, there has been substantial advances in the automated theorem proving area and cloud computing in the last decade. We believe that by combining these two technologies we can reduce the number of interactive proofs and thus allow industry to adopt formal techniques.

Motivation Summary *It is essential to demonstrate that safety-critical systems we design work in a correct and safe way. A common approach is to construct prototypes in early stages of the development and then gradually refine system by testing is insufficient. Formal methods is a desirable approach, however, we need better tools which can cope with large problems. In particular, we need to address the issue of interactive proofs to make formal methods more applicable.*

1.2 Thesis Contribution

In the previous section we emphasised the importance of formal modelling especially for validating safety-critical systems. We also discussed the challenges industry and academia face in verifying these complex systems. In particular, we focused on interactive proofs - the crux in adopting formal methods. A good tool support is necessary in order to apply formal techniques to complex and large projects [Abr07].

A substantial research effort has been put by academia and industry in the area of theorem proving. There are more than a dozen of different theorem provers which are different in many aspects, for example: performance, input language, logic. Hosting a collection of them would dramatically increase the success rate of proving a proof task. However, creating a link to each of them from the desired formal specification language can be a challenging task.

Doing proofs on the cloud opens possibilities that we believe were previously not explored, outside, perhaps, prover contests results. Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable

computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. Provers-as-a-service is a natural direction given that provers are CPU and memory intensive. At the same time, running a collection of (distinct) provers on the same conjecture is a trivial and fairly effective way to speed up proof given plentiful resources.

In a nutshell the thesis proposes to address the interactive proof problem by developing a new verification platform to help mechanise the proving process. The proposed verification technique relies on two key pillars: automated theorem proving and cloud technology. The main objectives of this study are outlined below:

Goal G_1 *To increase verification automation of the formal models by designing a new verification platform which exploits the state-of-the-art theorem proving and cloud technologies.*

Even though, the state-of-the-art theorem provers and cloud technology can dramatically reduce the numbers of interactive proofs they cannot ensure completely automate verification. The current state of technology requires some kind of interactive proof mechanism. Thus, in our work we propose to address one of the most common problems of interactive proofs - proof fragility and re-usability, G_2.

Goal G_2 *To make interactive proofs more generic and thus less fragile, and more reusable. The crux of the technique is to offer an engineer an opportunity to complete a proof by positing and proving a generic lemma that may be reused in the same or even another project.*

To evaluate our approach of doing proofs with our technique we have set another objective in this research - G_3. The thesis presents advantages and disadvantages of our technique by comparing developed verification method against established verification tactics.

Goal G_3 *To evaluate the new verification platform by comparing its performance against established provers.*

Lastly, it is important to note and distinguish personal contributions of this research as work was completed in collaboration with other two colleagues. However, I feel that it is important to present the complete picture of the tool. Hence, the thesis briefly discusses work were my contribution was trivial.

The cloud interface to our verification platform was developed by David who is a PhD student at Newcastle University with the interest in dependable software engineering and cloud computing.

Alexei Iliasov - Research Associate at Newcastle University - has a long experience in the field of formal methods. He has developed numerous tools and worked on several large projects. He has greatly contributed in developing and integrating verification plug-in.

I personally was responsible for developing proof tasks translator from Event-B to Why3 notation, and also, axiomatising the Event-B theory (Section 4.1 - 4.2). In Section 5 where we evaluated our technique I used our tool to prove Event-B models and deduce additional schematic lemmas.

1.3 Thesis Structure

The thesis is organised into six main chapters and following outline provides a summary of each chapter.

In **Chapter 1** contains the motivation behind this research by emphasising the importance of formal system analysis, verification and its automation. In the same chapter we outline thesis contributions and research objectives.

Chapter 2 contains an overview of research areas which are relevant to this work. We particularly focuses on the Event-B formal specification language which is the basis of this research. This chapter also presents the notion of theorem proving and the state-of-the-art tools.

The verification tool architecture is presented in **Chapter 3** which explains how we combine cloud and theorem proving technologies. The chapter also contains a high-level description of tool operation.

Chapter 4 describes the implementation of the architecture. It is split into four main work directions: sequent translation, set-theoretic operator axiomatisation, generic lemmas and hypothesis filtering.

In **Chapter 5** we discuss perceived advantages and disadvantages of doing proofs with our technique by proving several fairly well-known Event-B models. The chapter is split into two parts where we present results of automatic and interactive modes of the tool.

We discuss how this verification platform could be integrated and used in specific domains in **Chapter 6**. In the same chapter we draw research conclusions and outline possible future work directions.

Publications

Some of the work presented in this thesis as well as contributions to the research have been previously published or presented as follows:

- A. Iliasov, P. Stankaitis, D. Ebo Adjepon - Yamoah and A. Romanovsky, ”Rodin Platform Why3 Plug-In,” *In Proc. of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM and Z.* ABZ 2016, May 23 - 27, Linz, Austria. Springer, LNCS - 9675, Pp. 275 - 281.

- A. Iliasov, D. Ebo Adjepon - Yamoah, P. Stankaitis, and A. Romanovsky. ”Putting Provers on the Cloud,” *In the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.* PDP 2015. Work in progress. March 4 - 6, 2015. Turku, Finland.

- A. Iliasov, P. Stankaitis and D. Ebo Adjepon - Yamoah and A. Romanovsky. ”Proving Event-B models with reusable generic lemmas,” *In Proc. of 18th Formal Engineering Methods. ICFEM 2016.* November 14 - 18, Tokyo, Japan, 2016.

- A. Iliasov, P. Stankaitis and D. Ebo Adjepon - Yamoah. ”Event-B and Cloud Provers,” *In Proc. of the 22nd Workshop on Automated Reasoning.* ARW 2015, April 9 - 10, Birmingham, UK, 2015.

- A. Iliasov, P. Stankaitis and D. Ebo Adjepon - Yamoah and A. Romanovsky. ”A Rodin plug-in for constructing reusable schematic lemmas,” *In Proc. of 6th Rodin User and Developer Workshop.* May 23, Linz, Austria, 2016.

- A. Iliasov, P. Stankaitis and D. Ebo Adjepon - Yamoah. ”Event-B and Cloud Provers,” *In Proc. of 6th International Workshop on the use of AI in Formal Methods.* AI4FM 2015, September 1, Edinburgh, UK, 2015.

2 Formal Methods and Automated Theorem Proving

In this chapter we aim to give an overview of formal methods - a mathematical approach for a formal system specification and verification. The chapter discusses reasons behind poor applicability of formal methods in the past, major advancements and present problems.

In following sections we review few surveys on formal methods, present famous and successful industrial cases, and lastly, discuss and classify a number of popular formal method languages. In particular, we focus on the Event-B method and tool developed to implement it - the Rodin Platform. The final section discusses the paradigm of the theorem proving and overview state of the art automated theorem provers.

2.1 Formal Methods

A constantly increasing complexity of systems pose uncertainty whether a high level system reliability can be guaranteed. Rigorous system development methods are quintessential to evidently reduce a risk of a human error. In particular, for many modern applications which we classify as safety-critical a subtle human error may cause a tremendous damage. To reduce a potential risk certification standards have been issued which require providing a clear systems' correctness evidence to the certification authority.

The majority of engineering disciplines use mathematical techniques and tools to guarantee correctness and reliability of their developed systems. Perhaps, because of the software engineering nature, applying mathematical techniques to ensure correctness has been hardly the case. A more widely used approach in the software projects is the verification by a thorough testing. In contrast to other applications, a software can be easily modified at a fairly low cost which makes software application refinement through testing very attractive to industries. Moreover, the testing process is more intuitive to engineers, test cases can be automatically generated and the simulation part can be visuallised.

An alternative approach, formal methods, is a mathematically based approach for specification and verification of complex and safety-critical software and hardware systems. Because of a several popular misconceptions, outside complex and safety-critical system development formal methods have been hardly applied [Hal90; Abr10].

Historically popular opinion about formal methods is they are necessarily costly with an inadequate tool support, or simply too hard to use [CW96]. That was certainly the case in a past, however, a number of surveys and reports have shown a positive industrial feedback on integrating formal methods in the development process (e.g. [WLBF09; Hax10]).

Although, a universal feedback is encouraging, various industrial domains and companies report different opinions on applicability of formal methods, as there is still a need for a more generic and universal formal techniques and tools. Since, a testing contributes significantly to the software project cost, a very satisfying outcome of applying formal methods is reduced testing effort. In the past, due to inadequate tooling formal verification role in system validation was unknown. Testing will continue to remain essential [BH95] for a long time, as survey correspondents identify formal method tooling currently insufficient for commercial applications.

The formal methods survey by Woodcock et al. [WLBF09] showed that the majority of projects used formal techniques for specification and modelling. In designing a system one must specify a number of parameters, which define system behaviour, performance and constrains. Generally, using natural languages in system requirement or behavioural specification phase can introduce faults, as specification could be misinterpreted due to the language ambiguity. In contrary, specification phase can be also completed in a formal manner, system behaviour specifications and requirements become an useful mathematical artifact rather than redundant documents.

Formal methods offer an approach to specify system precisely via a mathematically defined syntax and semantics. It was demonstrated, spending extra effort in early specification phase can discover flaws and inconsistencies in the early design phase which can save a lot of money and time in testing. Throughout the years many modelling languages were developed to allow specification of sequential and concurrent systems, describe behavioural, performance and other properties. In fact, formal languages in themselves are different based on modelling principles, problem suitability or semantics. Although, generally formal methods could be classified into following categories: a transition-based (StateCharts [Har87]), a state-based (VDM [BJ78], B [Abr96]), a history-based (CSP [BHR84]) and a process-based (Petri Nets [Pet81]).

In choosing a suitable formal specification language many aspects need to be considered, for instance tool support and language expressiveness. However, only by choosing the most appropriate formalism for the particular system, business benefits can be maximised. A few formal method surveys we briefly reviewed, can work as potential industry guidelines in adopting formal methods.

A lot of industrial applications which incorporated formal methods in the design process only used it for system specification. Even though, formal specification can indicate potential faults early in development phase, a more substantial evidence is needed. The verification by a thorough testing remains a popular method to guarantee correctness of the system, however, for the most of systems an exhaustive testing is unachievable. On the contrary, the formal verification can provide a higher level of assurance, however, an applicability of formal verification is highly dependent on tools.

In essence there exist two formal verification techniques: theorem proving and model checking. The latter has been explored for a long time now and has been successfully applied in verification of hardware, software system specification and protocols. The core of the model checking is to construct a finite-state model and automatically, and exhaustively check whether the model meets requirements. In general, practical models consists of enormous number of states, therefore, to check all possible states efficient search algorithms are essential.

There are a lot of different model checking tools, for instance SPIN [Hol03], ProB [LB03] and NuSMV [CCGR00]. They support various modelling languages and are based on different logic's. Perhaps, the two biggest advantages of model checkers are counters-examples which can be used to refine the system and fairly short running time, typically minutes. However, even with powerful abstraction techniques the state-space explosion remains a big challenge. The alternative technique for automated verification is theorem proving (see Section 2.3).

Formal methods success highly depends on a academic community effort to improve and produce new methods and tools. The education also plays an important role, as training software engineers formal techniques can be fairly expensive, although, a one-time investment. A lot has changed in the last two decades, formal methods are getting increasingly accepted by industry. There are already formal languages and verification tools which have been applied in a large industrial project and withstand applicability criticism. In the following section we aim to overview several industrial cases and domains where formal methods were successfully applied.

2.1.1 Formal methods in industrial applications

Certification authorities increasingly recognise benefits of formal methods, and therefore, several standards (e.g. CENELEC EN 51028) highly recommend using formal techniques for software specification and verification. Even though, standards do not require using a specific method or language nor applying formal methods is mandatory in general, this recognition implies a growing importance of formal methods [FFM13].

A series of successful projects in the past pushed an acceptance of formal techniques. For instance, some early examples include, formally specifying transaction processing system [Hal99] and air traffic control systems [HK91; LHHRO91]. In particular, the railway sector has been a fruitful in formally specifying and verifying railway signalling systems. Perhaps the most notable examples was the use of the B-method in major railway projects, for instance a Paris metro line 14. The technique used for this particular project included formalisation of requirements, construction of abstract model and its refinement to a concrete design and verification of safety properties. Indeed, the latter exposed a number of bugs, and thus, testing costs were minimised. Even though, verification process comprised of a large number of interactive proofs, report states [BBFM99] that proving process was reasonable. Furthermore, over three development years project itself forced the evolution of B toolkit which was identified as particularly satisfying outcome. Subsequently, similar formal development process was applied and in different countries and lines, for instance New York Canarsie line, Barcelona subway.

Nonetheless, the transport is not the only domain where formal methods were successfully applied for a large scale application. The financial sector is known for its stringent system requirements which mandate using formal methods specify and verify security policies. In the late nineties, an electronic cash system, Mondex, achieved the highest certification level by integrating formal methods in the development process. As a result, testing revealed no implementation bugs in the system parts where formal methods have been used. The electronic cash system was abstracted in Z method [Spi89] and manually proved to be secure.

In general large-scale industrial projects which successfully applied formal techniques in an application development report that formal method not only increase the quality of the product, but are a cost and time effective. Formal methods reports (e.g. [WLBF09; Hax10]) show that formal techniques are applied predominantly in transport and finance sectors. However, a lot more application domains have been successfully adopting formal methods, for instance medical, defence, nuclear and so forth.

2.2 Event-B Method

In this section we discuss the Event-B modelling language [Abr10]. To begin with, we concisely review refinement-based Event-B model development approach and types of verification conditions one needs to prove to demonstrate model correctness. Furthermore, we present the structure of the Event-B model and modelling notation.

The Event-B mathematical language used in the system development and analysis is an evolution of the Classical B method [Abr96] and Action Systems [BS89]. It offers a fairly high-level mathematical language based on First Order Logic and Zermelo-Fraenkel set theory and an economical yet expressive modelling notation. Event-B belongs to a family of state-based modelling languages that represent a design as a combination of state and transitions. In simple terms a state of a discrete system is just a collection of variables and constants where as the transition is a guarded transformation which modifies variables.

Modelling complex systems require composing a large number of these states and discrete transitions. Therefore the incremental or a top-down design approach is the most suitable in designing such systems. A cornerstone of the Event-B method is the step-wise development that facilitates a gradual design of a system implementation through a number of correctness-preserving refinement steps. The Event-B model development starts with a creation of a very abstract specification. The concrete Event-B model is completed when all requirements and specifications are covered in the model.

A sufficiently detailed model can be further refined by using Event-B mechanism of a vertical refinement. The vertical refinement process is used to transform a model from which an executional code can be generated. The Rodin platform which implements Event-B supports code generation to a number of programming languages.

However, constructing a model through a gradual refinement can reduce model complexity just to a certain limited extent. To dramatically minimise the complexity of the model one has to partition a system into smaller components via a systematic decomposition method. A kind of structural decomposition in Event-B can be done on a machine and event level [DB09]. Furthermore, there is a fine interplay between complexity of the model and automatically discharging proof obligations, so additional time spent in model specification can reduce the interactive proof effort.

The Event-B model is made of two key components which describe dynamic part of the model (*machine*) and static (*context*). The context contains modeller declared constants and associated axioms which can be made visible in the *machines* with `sees` function. The dynamic part of the model is described by variables (v) which are constrained by invariants $I(v)$ and initialised by action - $R(c, s, v')$.

```
machine M
  sees Context
  variables v
  invariant I(c, s, v)
  initialisation R(c, s, v')
  events
     E1  =  any vl where g(c, s, vl, v) then S(c, s, vl, v, v') end
     ...
end
```

Fig. 2.1: Event-B machine structure.

Variables are transformed by actions which are part of events. A modeller may use predicate guards to denote when event is triggered. To ensure a deadlock free system one must prove a deadlock freedom rule, in other words, at least of one the guards has to be enabled at any time. However, some applications permit deadlocks. A collection of contexts and machines which may refine each other compose an Event-B model. The abstract form of an Event-B model is shown in Figure 2.1.

Specifying a model is not sufficient, one must provide evidence about the correctness of the model. Event-B is a proof driven specification language where model correctness is demonstrated by generating and discharging *proof obligations* - theorems in the first-order logic. The model is considered to be correct when all proof obligations are discharged. In addition to deadlock freedom rule there are several other rules of proof obligations which needs to be verified. In what follows we overview few essential rules.

A well-definedness rule for expressions is there to ensure that a potentially ill-defined expressions are in fact well-defined. A typical example is function application expression, function application $f(E)$ is well-defined iff E is in the domain of the function f and f is not a relation. A further overview on well-definedness with a complete list of Event-B operator WD conditions as well as discussion on static and dynamic checking in Métayer et al. [MV09].

For further reading - Modelling in Event-B (Chapter 5), Jean-Raymond Abrial (2010)

In developing a Event-B model one must ensure that variable transformation in events satisfies invariants also known as the *invariant preservation rule* (Eq. 2.1). In an equation below we represent $A(s, c)$ as collection of axioms and theorems, $I(s, c, v)$ invariants, $G(s, c, v, x)$ guards of the event and $BA(s, c, v, v', x)$ before-after predicate of the event.

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, v', x) \Rightarrow I(s, c, v') \tag{2.1}$$

There are several other rules of proof obligations, for instance to demonstrate that refinement process did not refute one needs to prove refinement relation verification conditions of Event-B models or a feasibility rule which ensures non-deterministic action feasibility.

2.2.1 Rodin platform

The practicality and applicability of formal methods heavily relies on the tool support. In order to make Event-B and its toolkit appealing and competitive in an industrial setting there was a concerted effort, funded by a succession of EU research projects [BJRT06; RT13]. One of the main outcomes of these projects was the Rodin platform which offers a fairly capable development and proof support for the Event-B specification language.

The Rodin platform provides two separate environments for modelling and verification of Event-B models. The modelling environment of the tool provides a user-friendly graphical interface to construct and edit Event-B models.

The platform also provides an interactive proving environment which offers a number of automated theorem provers and several simplification rules. These can be mixed to create various verification tactics. The Rodin toolkit can be equipped with several automated theorem provers, for example nPP, PP and ML. In case of auto-tactic fails to prove, proof can be completed interactively.

Voisin and Abrial [VA14] emphasise the openness of Event-B modelling approach which was also reflected in developing the Rodin Platform. It was built on a Eclipse framework foundation which allows tool to be easily customised and independent.

Throughout the years several useful tools were developed to facilitate modelling and verification activities. The Event-B notation is not closed and defining additional operators is permitted. Theory plug-in [BM13] developed at Southampton University

allows explicitly or axiomatically defining new operators as well as adding new theorems, data-types and simplification rules.

In recent years there has been major advancements in the automated theorem proving field. Largely in the developing new calculi and first-order logic solvers. In order to complement existing verification techniques in Rodin some important work has been done by Déharbe et al. to provide an interface to SMT solvers [DFGV12]. The latest version of SMT plug-in contains several SMT solvers and has shown a great potential, however, further improvements are needed to support a complete set theory.

At the moment, one of the biggest problems in automated theorem proving area is dismissing irrelevant hypothesis. The Rodin platform attempts to reduce the state-space for the provers with relevant hypothesis filtering techniques. The idea is simple - to dismiss irrelevant properties before automatic prover is even called. In general, there are several techniques for relevant hypothesis filtering, for example methods based on abstraction or history-based technique which inspects previously completed proofs (adaptive). A popular yet rather simplistic method is to use a syntactic filtering technique which look for specific symbols in hypothesis and goal. The Rodin platform has several syntactic hypothesis filters available which come with Relevance Filter Plug-in [Röd10]. A few other important Rodin plug-ins include an extension for code generation [ERB12; MS11], model-checker ProB [LB08] and UML front end for graphical Event-B modelling [SB06]. A complete list of Rodin extensions can be found on the Event-B[1] page.

The development platform which now has been deployed for more than a decade has an active community which continues improving the platform and developing tool extensions. In addition, a biennial Rodin User and Developer Workshop has been running for several years now.

2.3 Automated Theorem Proving

So far in the thesis we discussed two approaches for the correctness validation a thorough testing and a formal approach - model checking. Even though, both methods are widely used in the industry one and the other suffer from state space explosion. A theorem proving verification method, on the other hand, has no such an issue. A mathematical proof not only can provide more assurance of the system properties, but also provide more insight about the system. In this section we aim to concisely discuss the automated theorem proving paradigm, advantages and

[1] Event-B Wiki-page, `http://wiki.event-b.org/index.php/Rodin_Plug-ins`

disadvantages applying this technique and overview the state of the art automated theorem provers.

Automated theorem provers for a first-order logic have been used now for several decades to prove theorems and to do it automatically. It has been shown that to prove interesting and complicated mathematical problems is hardly feasible with automated theorem provers [Fit96]. On the other hand, a model proof obligations are less complex and fundamentally could be proved by hand, however, there are few important reasons why this is not acceptable for a majority of computer system verification projects.

Large scale applications models contain a huge amount of proof obligations. Proving all of them manually is simply impractical and potentially error prone. For instance, in verifying electronic cash system Mondex to reduce project costs a 200-page proof was done manually. Other projects where formal verification was used to provide required confidence report that even with automated theorem provers interactive proofs costed a lot of effort to discharge remaining proof obligations (see [Abr07; RT13]). In the industrial setting domain engineers are very rarely capable of carrying out a mathematical proof and their training can be very expensive. Therefore, a much preferred option is to mechanise the theorem proving exercise.

The automated reasoning area has made a great progress in automating the verification process over the last few decades. Particularly in developing new proof search heuristics and theorem provers for first-order logic. In general, virtually all practical proof search approaches in automated theorem proving are based on the resolution principle by Robinson [Rob65].

First of all, the resolution method transforms a conjecture (propositional formula) to be proved into a standardised conjunctive normal form (CNF). A conjunctive normal formula is made of clauses (logical expressions) where a clause is a set of disjunctive atomic literals. For instance, a following propositional formula $(X_1 \wedge X_2) \vee X_3$ can be transformed into a CNF standard $(X_1 \vee X_2) \wedge (X_1 \vee X_3)$ by applying simple logical rules. In theory, any propositional formula can be transformed into a clausal normal form, however, the problem of such a transformation is an exponential increase of formula size. Hence, instead of modifying a formula to be logically equivalent it is enough to preserve a satisfiability. A number of algorithms have been developed to check satisfiability of propositional formulas in conjunctive normal form, see DPLL algorithm by David et al. [DLL62]. Yet, a slightly more complicated process is a transformation of first order logic formulas in clausal normal form, since formula has to be freed of universal quantifiers, Nonnengart et al. [NW01].

In principle the resolution method uses a proof by contradiction tactic, so one needs to negate the conjecture and place it in conjunctive normal form formula as one of the clauses. Then, clauses are combined in pairs to construct new clauses and process in continued until an empty clause is reached. An empty clause indicates about the inconsistency in the axiomatic system which tells that a conjecture is indeed a theorem. Nonetheless, there exist other theorem proving methods, like natural deduction, sequent calculus or a truth tree based method - analytic tableaux [Smu95]. However, due to their expressiveness which increases the search space they are not particularly suitable for the automated theorem proving method, discussed by Bridge [Bri10].

Satisfiability Modulo Theories, a more general form of Boolean Satisfiability Problem, is a decision problem of classical first-order formulas which contain predicate symbols instead of binary variables. A number of theorem provers have been developed to solve Satisfiability Modulo Theories problems.

There was a concerted international initiative to standartise SMT solvers; the majority of solvers now support a standard SMT-LIB input notation [BFT10]. Furthermore, annual automated theorem proving competitions has had an significant impact on developing and improving verification tools, refer to competitions CASC [SS06] and SMT-COMP [BDDMOS13]. Competitions, also revealed an interest in automated reasoning area not only from the academic community but the industry too, see Moura et al. [MB08] and lastly they also work as guidelines for future improvements and current problems.

In the following section, due to significance to our research we aim to overview a particular tool called - Why3 which brings a large number of automated theorem provers under one palatable platform.

2.3.1 Why3 for deductive reasoning

In the last two decades, a large number of automated theorem provers have been developed which are different in many aspects. Recently some important work has been done to bring a large number provers under the roof of a common, versatile notation - the Why3 verification platform [BFMP11]. Why3 offers a common interface to over a dozen of automated provers; it also has its own high-level specification notation to reason about software correctness but in this section we focus on the former.

The Why3 tool has been used as a link between system development platforms and automated theorem provers. In a paper by Dejanira et al. [AIER14] an approach was presented for a verification of high-level control systems' properties in `Simulink`

modelling tool using the Why3 platform. In the other work the tool was used a back-end verification plug-in for the CheckSPS platform for reasoning about structure-preserving signature schemes which are useful for cryptographic operations [DR14]. The Why3 verification system was also used to interface with a number of automated theorem provers for SUPPL programming language, by Dockins and Tolmach [DT14]. A language designed for event driven architectures supports static analysis and can discover inconsistent policies with Why3. In all these projects, Why3 provided an interface to back-end provers where research groups developed a translation tool from their specific platform.

Why3 operates on a first-order logic with polymorphic types with some extensions. Furthermore, it understands recursive definitions, inductive predicates and algebraic data types. In addition to built-in types integers, real numbers and polymorphic tuples one can declare enumerated types or simply a non interpreted type, for instance:

```
type graph = Vertex | Node
type graph
```

Why3 input format is a collection of small units - theories which contain declarations of functions and predicates, types, axioms and lemmas, and goals. The theory can import already existing declarations from other theory files or else clone a library which creates a local copies by using functionality showed below:

```
use   import file.Theory
clone import file.Theory
```

A standard Why3 library provides a number of built-in libraries including theories on integers, graphs, sequences and many others. Nonetheless, one can construct a library using `function` and `predicate` declarations. Functions can work over a variety of types including a polymorphic type, a general function format is as follows:

```
function name type : type
```

Predicates and functions can be explicitly defined recursively using a `match` operator or inductively by satisfying a set of clauses. In addition, other Why3 functionality like pattern matching or `let` and conditional expressions can be used to construct functions which are then transformed in a classical first order logic. The main

objective in reasoning with the Why3 tool is to prove a goal which can be stated as shown in declaration below.

```
goal name: identifiers : type. formula
```

In order to help to prove a goal axioms and lemmas can be included in the Why3 theory.

```
axiom name: identifiers : type. formula
lemma name: identifiers : type. formula
```

At the basic level Why3 is a sequent processing tool which uses configurations files (drivers) to complete a series theory transformations. To begin with, Why3 constructs a flat proof task meaning that a transformed theory does not import additional libraries. The specific prover configuration files detail pretty-printing function which corresponds to a particular input notation and logic, for instance, a number of provers do not support polymorphic types or conditional expressions which are permitted. Why3 is a flexible verification platform, these configurations files can be manually modified or added to Why3 so newly developed provers can be supported.

3 Verification Tool Architecture and Proof Scenarios

In the previous chapters we argued about the importance of formal methods and verification automation. In particular, we focused on the Event-B specification language and automated theorem proving technique. This chapter aims to describe an architecture which exploits mentioned techniques, and which will be used to develop a verification plug-in.

To begin with, we concisely overview the Event-B/Rodin technology and explain how it could benefit from the Why3 umbrella prover. Then, we describe the overall design and operation of the plug-in. In the last section, we present the cloud service side of the tool where we defined the notion of a proof scenario and explain how parallelism, and elasticity of cloud technology can be exploited for proofs.

3.1 Verification Plug-in Architecture

The following section proposes an adaptable and scalable verification architecture which effectively could support a variety of formal specification languages. However, we focus on how this architecture could be fitted with a popular formal modelling notation - Event-B. It has been long recognised that the Event-B/Rodin technology may significantly benefit from an interface between Event-B and TPTP theorem provers. This thesis proposes the verification plug-in which hosts a collection provers on a cloud and provides an interface to them.

The architecture is made of verification and modelling environments. In the latter, an environment provides an interface to develop a model, and furthermore, to automatically generate verification conditions. In the instance of Event-B, the modelling environment is represented by the Rodin platform where theorems in the first-order logic are generated. Whereas, the verification side of the architecture is an extension (e.g. a Rodin plug-in) of the modelling environment which hosts locally or remotely a number of automated theorem provers.

At the core of the verification environment we propose to use a popular umbrella prover - Why3, to facilitate the translation of proof obligations from Event-B to TPTP theorem provers. Why3 offers a common interface to over a dozen of automated

provers and has a palatable input, the tools also allows constructing and importing additional libraries which we exploit to the support Event-B notation. Figure 3.1 visually illustrates the proposed verification plug-in architecture for the Rodin platform.

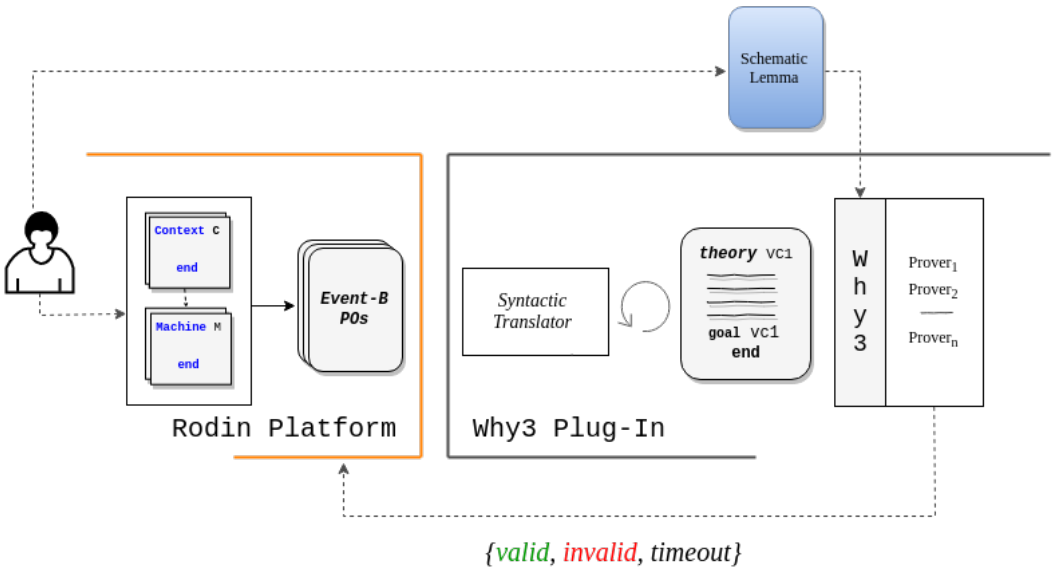

Fig. 3.1: Architecture of the verification plug-in

The verification side can be further decomposed into following elements: lightweight syntactic translator, Why3 library of the Event-B language, lemmata filter and interface for schematic lemmas.

In addition to the syntactic translation of the proof obligation, one must also formalise the Event-B modelling theory in Why3. The umbrella prover does not rely on a small proof kernel and allows one to make axiomatic definitions. It is a much quicker way to define an embedding of a logic but there is always a danger of making it unsound. In a simpler case, an unsound axiomatisation may be detected by proving of a tautological falsity but there are more intricate situations where unsound definitions show up only in specific circumstances.

We define each Event-B operator in a separate theory and give the bare minimum axiomatic definition that must be checked by hand. Furthermore, we constructed and proved a fairly long list of supporting conditions. These are deposited in a separate theory file. A more detailed explanation of the syntactic translation and set-theoretic operator axiomatisation is presented in Sections 4.1 - 4.2.

Previously, we set an objective to offer user an opportunity to complete a proof by positing and proving a generic lemma. In our view a generic lemma would not only solve the problem of insufficient axiomatisation, but would further address the interactive proof problem.

The verification model contains an interface for entering a generic lemma and a filtering mechanism to rank lemmata. In Figure 3.2 we graphically illustrate the translated proof task transformation process.

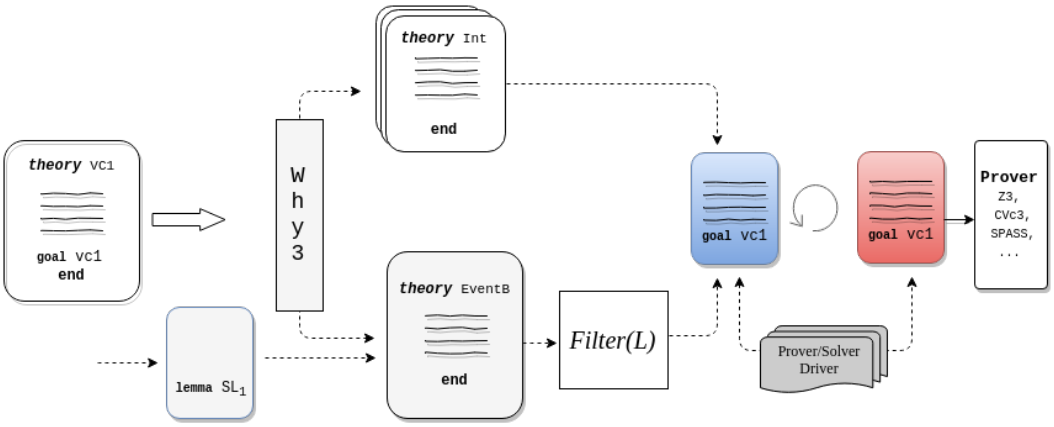

Fig. 3.2: The proof task transformation by Why3

The Why3 tool completes a series of transformations driven by a prover configuration files to transform proof task to a specific prover format. Furthermore, at this stage the plug-in also detects suitable schematic lemmas from the supporting theory library. The final proof task is passed to the specific theorem provers determined by a proof scenario. In the following, we summarise the main steps of the Rodin - Why3 verification tool chain.

- A user develops a Event-B model;
- The Rodin Platform automatically generates proofs obligations (POs);
- A user optionally selects the operation mode, timeout and verification scenario;
- Proof obligations are individually parsed and translated into the Why3 notation;
- Translated POs are processed by Why3 to include required libraries and schematic lemmata;
- Processed POs are further transformed according to prover/solver driver;
- Verification conditions are forwarded to provers defined by a proof scenario;
- Results are collated and parsed back to the user;

The verification perspective could function either in a local mode when all the provers are installed locally, or remotely as a cloud service. In the following section

we provide a more concise description of the cloud service side of the verification tool.

3.2 Cloud and Prover Scenarios

Doing proofs on the cloud opens possibilities that we believe were previously not explored, outside, perhaps, prover contests results. Given that provers are CPU and memory intensive and there is a great potential for exploiting parallel processing, from the outset we were aiming at provers-as-a-service cloud architecture. Indeed, running a collection of (distinct) provers on the same conjecture is a trivial and fairly effective way to speed up proofs given plentiful resources.

The usability perception of interactive modelling methods such as Event-B is sensitive to peak performance when a burst of activity (new invariant) is followed by a relatively long period of idling (modeller thinking and entering model). The cloud paradigm where only the actual CPU time is rented, seems well suited to such scenario. Also, the cloud's feature of scalability plays a critical role in this situation. Virtualization which is a key concept of cloud computing enables the installation of multiple virtual machines on the same physical machine. It supports scaling through load balancing. Here, a service running on a provisioned virtual machine scales either up or down to handle varying amounts of requests using a live migration process.

The cloud service accepts as inputs sequents $\mathcal{S}$ of the form: (p, t) where p defines a proof scenario stipulating which provers need to be run and t is a Why3 theory containing a single goal.

The server executes a proof scenario p to obtain a proof result. A proof scenario is a function from an input sequent to a proof result: $q \in \mathcal{S} \rightarrow \{\mathsf{unknown}, \mathsf{valid}, \mathsf{invalid}\}$ and is defined via the following proof scenario primitives:

$$
\begin{array}{lcll}
p(t) & := & \mathsf{pr}(t) & \text{(prover call, positive)} \\
 & | & \mathsf{pr}(\neg t) & \text{(prover call, negative)} \\
 & | & p(t) \triangleright w & \text{(deadline)}, w \in \mathbb{R}_+ \\
 & | & p_1(t) \wedge p_2(t) & \text{(\textit{and} composition)} \\
 & | & p_1(t) \vee p_2(t) & \text{(\textit{or} composition)} \\
 & | & \neg p(t) & \text{(result negation)}
\end{array}
\tag{3.1}
$$

The negation operator $\neg$ on proof results turns valid into invalid, invalid into valid and does not affect unknown and failed.

The *or* composition $(p_1 \vee p_2)(t)$ is opportunistic: it may return any result r such that $r \in \{p_1(t), p_2(t)\} \setminus \{\mathsf{unknown}\}$, and when no such result exists, returns unknown. The *and* composition $(p_1 \wedge p_2)(t)$ evaluates $\max(\{p_1(t), p_2(t)\})$ where unknown $<$ valid $<$ invalid.

The compositions are distributive and commutative so that provers invocations may be scheduled rather flexibly or invoked at the same time. In practical terms, the *or* composition runs until any prover returns a definite results and the *and* composition runs all the provers until it sees invalid result. The multiplicity of (independently developed) back-end verification tools may be relied on to increase the confidence in a proof result by applying adjudicating on the results of prover calls:

$$SA(t) \equiv \mathsf{pr}_1(t) \wedge \mathsf{pr}_2(t) \wedge \mathsf{pr}_3(t) \wedge \ldots \tag{3.2}$$

An important case is proving both positive and negative forms of an input sequent: $\mathsf{pr}_1(t) \wedge \neg\mathsf{pr}_1(\neg t)$. Negation may also be employed opportunistically with the parallel composition: $\mathsf{pr}_1(t) \vee \neg\mathsf{pr}_1(\neg t)$.

Provers may be run with a timeout. A practical example is to run a less capable but often faster prover in parallel with a slower prover:

$$\mathsf{pr}_1(t) \triangleright w_1 \vee \mathsf{pr}_2(t) \triangleright (w_1 + w_2), w_1, w_2 \in \mathbb{R}_+ \tag{3.3}$$

An efficient implementations of both sequential and parallel compositions must rely on concurrent invocation of some or all of the composed prover calls.

There is a fairly large number of independently designed provers, they complement each other well and the best results are achieved when several provers are working in parallel on the same verification condition. However, this is fairly expensive. Using, say, 12 provers, requires a 12-core machine to to run them at the same time. Typically, a single change in a model or code generates several (sometimes hundreds) verification conditions and these must be processes as quickly as possible. A normal desktop or workstation cannot cope with such a load efficiently.

Luckily, the problem is trivial to parallelise: each verification condition is completely self-contained and can be processed independently of others. A prover operates independently of other provers. Thus the typical verification workload is a convenient problem for a distributed server farm or a cloud.

4 Verification Tool Implementation

The previous chapter proposed a new verification plug-in for the Rodin platform. We abstractly explained how a such system would operate and could be implemented. This chapter is partitioned into four main sections where each section describes a particular technical implementation challenge.

To begin with, we discuss the syntactic translation of a proof sequent and a formalisation of the Event-B theory. We use concrete examples to explain technical details and challenges in implementing the plug-in. In Sections 4.3 - 4.4 we discuss the problem of incomplete axiomatisation and non reusable interactive proofs. For that reason we propose a schematic lemma concept and discuss how it can be used together with filtering to increase verification automation.

4.1 Syntactic Proof Sequent Translation

The objective of the syntactic proof sequent translator is to transform Event-B proof obligations so they would be compatible with Why3. So that for example the formula in Event-B $f \setminus \{t\}$ would become $(\texttt{diff}\ f\ (\texttt{singleton}\ t))$ in Why3. The translator pretty-prints an abstract syntax tree of an Event-B formula as an symbolic expression with a static mapping of Event-B operator names.

At first a seemingly trivial development exercise required a lot more effort because of several semantical differences particularly with respect to treatment of types. The Why3 language has a different treatment of types - type variables are explicit and are separate from the notion of a set - hence every carrier set in Event-B is defined twice in Why3: as a type variable and as a maximal set. For instance, carrier set CORES and enumerated set STATUS are translated into:

```
type tp_CORES
type tp_STATUS = T_ON | T_OFF

constant id_STATUS : (set tp_STATUS)
constant id_CORES : (set tp_CORES)

axiom hyp1 :(maxset id_STATUS)
axiom hyp2 :(maxset id_CORES)
```

There are two non-trivial mappings: the folding of left- or right- associative multi-operators into equivalent binary forms, and the detection of enumerated set definitions (a native, algebraic definition of enumerated sets significantly improves prover performance). Free identifiers occurring in a sequent become constants of a Why3 theory; hypotheses are theory axioms and the sequent goal is mapped into a theory goal.

```
public void translate(Expression expression)
                                          throws TranslationException {
    %match (Expression expression) {

      Card (X)              -> { prefixUnaryExpression("card", `X);}
      Ran (X)               -> { prefixUnaryExpression("ran", `X); }
      Pow1 (X)              -> { prefixUnaryExpression("pow1", `X);}

}

private void prefixUnaryExpression(String op, Expression a)
                                           throws TranslationException {

                enter(op);
                sb.append("(");
                sb.append(op);
                sb.append("␣");
                translate(a);
                sb.append(")");
                leave();
  }
```

Listing 4.1: Expression translation function: pattern selection

The syntactic translator was written in Tom/Java programming languages and simply pretty-prints an Event-B formula as an symbolic expression. Because of its suitability for defining transformations we have chosen a Tom programming language for pattern matching. In the research we used the *match* Tom pattern matching constructor which can be viewed as a more classic switch/case operator which consists of two elements, pattern selection and action. The primitive match simply chooses the first pattern that matches a tree-based structure subject and triggers an action.

In Listings 4.1 we provide the example for translating of unary expressions. Once, a unary expression pattern is matched in the translate method, an action prefix unary expression is triggered which simply constructs a string of the form (operator a). The remaining operators are translated similarly, however, some required a more complex string constructing methods, particularly quantifiers and set comprehension.

4.2 Event-B Theory Formalisation in Why3

The formalisation of a complex mathematical language like the one of Event-B is likely to be an ever open problem. Therefore, the most of the translation effort goes into constructing and fine-tuning of Event-B support theories in Why3. A particularly challenging objective is to construct an optimal axiomatisation which would not significantly increase the state-space for the prover, yet give sufficient support to prove something.

In developing such theories it is easy to introduce an error which leads to an inconsistent theory, and therefore, anything can be proven. Hence, in addition to optimality criteria it is essential to guarantee the soundness of the theory. In one of the first attempts to define the Event-B theory we indeed developed an inconsistent formal system. We had several cases where a missing finiteness statement or use of bi-implication instead of implication operator resulted in contradicting axiom, and thus, an unsound theory.

Mentré et al. [MMFA12] in their work on the B-method verification plug-in had a similar concern. For that reason they applied the interactive prover, Coq, to cross-check axioms and avoid inconsistencies in their theory. Nonetheless, this approach dramatically increases the time and effort in defining additional operators. In order to guarantee the soundness we provided just a bare minimum of axiomatic definitions in which we had high confidence. A simple, although, a slightly naive and less assuring solution we used was to try and prove a contradicting dummy-lemma with our axiomatic system.

Theory Structure. The Event-B mathematical language can be structured in *expressions* and *predicates*, or even, further decomposed into smaller operator groups, as discussed by Métayer and Voisin [MV09]. Besides the theory soundness, previous experiments have shown that the structure of the operator library is not trivial. The Event-B language library evolved from a homogeneous Why3 theory file which contained all Event-B operators to a library where each operator is defined in a separate Why3 theory. A split library is more readable, and furthermore, simplifies implementation of filtering mechanism. But most important it reduces the state-space for provers since only required definitions would be imported.

In order to analyse the library structure of Event-B language defined in Why3 we constructed a connectivity graph (see Figure 4.1). The graph illustrates how individual operator theories are linked together. It is possible to construct such a graph by tracing what libraries are included with a `use import` command. The size of the node indicates how often an operator theory has been used in a defining other Event-B constructs.

Constructing connectivity graphs in developing such a theory can provide a vital information particularly for deducing weaknesses in the axiomatisation. The projection in the figure below shows how the membership operator theory is linked with other theories.

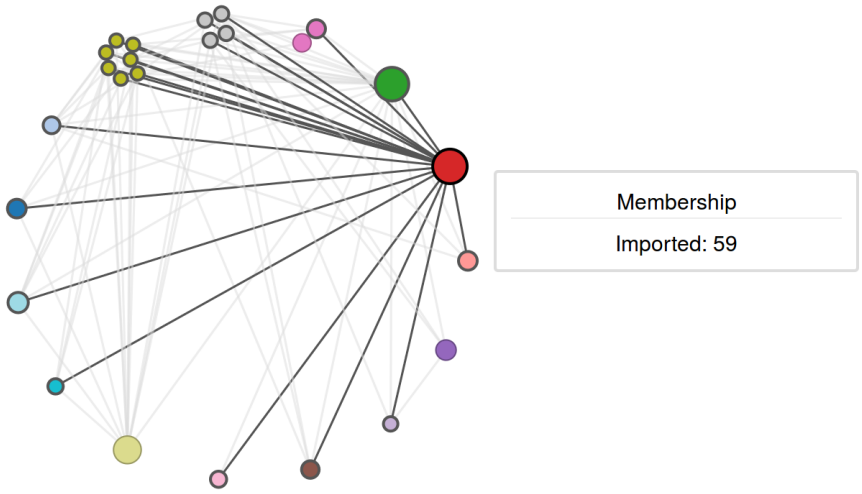

Fig. 4.1: Connectivity graph of the Event-B theory

Axiomatic System. There are several ways one might define an expression or a predicate. Why3 allows to declare an operator logic recursively and predicates inductively, or else, use the pattern matching technique for constructing operator definitions. A common approach is making use of Why3 functions `match` and `inductive` where user is required to define a number of clauses.

However, in defining operators we used an approach which was previously partially used in developing the SMT plug-in [MMFA12]. Instead of specifying clauses using inductive or pattern matching, we axiomatically defined operators using `axiom` and `lemma` declarations. A lot of Event-B operators were defined with respect to the operator - membership in a form of set equivalence.

In the following subsections we discuss few examples of Event-B operators formalised in Why3. We discuss two approaches in which an operator can be defined. The first option which is presented in Section 4.2.1 uses an inductive approach, whereas the operator presented in Section 4.3 is defined *directly*. We also discuss the idea of the support theory where we deposit additional support conditions.

4.2.1 Cardinality expression

This section presents a concrete example of an Event-B operator defined in Why3 - cardinality. The cardinality operator is an unary expression which simply takes an expression and produces another expression (integers and booleans are particular scalar expressions). In a mathematical sense, a cardinality is simply size of set (number of elements).

First of all, using the `use import` function we imported essential libraries which indicate that the theory uses symbols from following theories: Set, Finite, Membership and Int. The Finite library is needed as cardinality in the Event-B language is only well-defined for finite sets. The remaining libraries are normally imported in the later stages depending on axioms used to define an operator.

```
use import Set
use import Finite
use import Int
use import Membership
use import EmptySet
use import Singleton
use import ElementAddition
```

Listing 4.2: Why3 theory of set size operator

To model a cardinality operator we use a Why3 constructor - `function` which requires specifying operators name and types. In our theory cardinality works on polymorphic set type (*set 'a*) and returns an integer value - int (see below Listing 4.3).

```
function card (set 'a) : int
```

Listing 4.3: Why3 theory of set size operator

In general there are many ways one might define an operator, for instance a bijective function and an existential quantifier could be used to define a cardinality, however we chose an inductive method, shown in Listing 4.4.

Firstly, we establish the lower boundary of the operator. Then, we axiomatically define values of cardinality in special cases, for example axiom card_def2 specifies the set size of a single element set (*singleton*) or axiom defining set size of an empty set. In the final axiom we state the incremental part of the inductive approach.

```
axiom card_def0:
forall s: set 'a. (finite s) → card s ≥ 0

axiom card_def1:
forall s: set 'a.
    ((finite s) ∧ is_empty s) → (card s) = 0

axiom card_def2:
forall x : 'a.
    card (singleton x) = 1

axiom card_def3:
    forall x : 'a, s : set 'a.
    (finite s) ∧ not (mem x s) → card (add x s) = 1 + card s
```

Listing 4.4: Why3 theory of set size operator

In our opinion, the developed cardinality theory should be sufficient to prove some properties and yet not overwhelm provers. Although, it is the soundness that must be ensured in developing new theories. Therefore, we defined the operator in the most minimal still what we believe in a sound and sufficient way. Nonetheless, the only way to demonstrate this is by carrying out and analysing a number of proof conditions.

Analysing undischarged proof obligations of different Event-B models was a next step in the axiomatization process. It was noticed that a lot of undischarged goals required fairly simple, but very model-specific lemmas. Therefore, discovered missing lemmas had a high potential to be redundant in another context. On the other hand, we did not want to leave out newly discovered and potentially useful lemmas.

Furthermore, after early experiments with the Why3 tool was apparent that different provers prefer differing styles of operator definitions: some perform better with an inductive style (the size of an empty set is zero, adding one element to a set increases its size by one), while others prefer regress to already known concepts (here exists a bijection such that ...). Since we do not know how to define an optimal axiomatization, even for any one given prover, we provide basic axiomatisation and we offer an open translator with which a user may define, with as many cross-checks as practically reasonable, a custom embedding of Event-B into Why3.

Therefore, it was decided to leave the problem of supporting axioms partially open. Our goal was to provide just the basic definitions and support for Event-B operators, like shown in the cardinality example. Additionally discovered lemmas would be placed into separate theories with *TheoryName*_support and lemma_*LemmaName*

extensions to facilitate filtering. Listing 4.5 shows a supporting theory for the previously discussed cardinality operator.

```
theory Cardinality_support
  use import Set
  use import Cardinality
  use import EmptySet
  use import ElementAddition
  use import Finite
  ...
  use import ProperSubset
  use import Singleton
  use import Union

  lemma lemma_card_def7:
    forall s, t: set 'a.
      (finite t ∧ subset s t) → (card s) ≤ (card t)

  ...

  lemma lemma_card_def15:
    forall s t : set 'a, x : 'a.
      (not mem x s) →
              card (cprod (union s (singleton x)) t) =
              card (cprod s t) + (card t)

  lemma lemma_card_def16:
    forall s : set 'a, t : set 'b.
      card (cprod s t) = (card s) * (card t)


end
```

Listing 4.5: Why3 support theory of set size operator

4.2.2 Set comprehension expression

In this section we present a slightly more complicated operator - set comprehension. The function of set comprehension operator is to construct a new set whose elements satisfy a predicate *P* and are in a domain of some *F*. The Event-B method supports few forms of set comprehension. In our case we use the general case of the set comprehension $\{x \cdot P \mid F\}$ where we denote P as a predicate and F as an expression. For instance, using our definition of set builder one can construct a set of all even numbers $\{n \cdot P(n) \mid F(n)\} \Rightarrow \{n \cdot n \in \mathbb{N} \mid 2n\}$.

The complex part in formalising and translating a set comprehension was the use of the lambda abstraction. Lambda abstraction is a mathematical method which can be used to define and call functions. In the Why3 language lambda abstraction can be expressed with a backslash symbol.

In the listing below (Listing 4.6) we illustrate how a simple addition function can be abstracted, called and checked with lambda abstraction in Why3.

```
goal g0:
  (\ z:(int, int) . (let a, b = z in a + b)) (1, 2) = 3
```

Listing 4.6: Example of lambda abstraction in Why3

To define the Event-B set comprehension - *bsetc*, we used the built Why3 function, comprehension (see Listing 4.7). As an input of the function it uses a polymorphic high order predicate and function, and returns a polymorphic set - *set b'*. The function is axiomatised giving a *direct* definition through comprehension function and existential quantifier `exists` using lambda abstract.

```
theory SetComprehension

  use import Set
  use import Membership
  use HighOrd as HO

  function comprehension (p: HO.pred 'a) : set 'a

  axiom comprehension_def:
    forall p: HO.pred 'a.
          forall x: 'a. mem x (comprehension p) ↔ p x

  function bsetc  (p: HO.pred 'a) (f: HO.func 'a 'b) : set 'b =
              comprehension (\ y: 'b. exists x: 'a. p x ∧ y = f x)
end
```

Listing 4.7: Set comprehension model in Why3

In Listing 4.8 we show how the set builder expression $\{x, x_1 \cdot x + x_1 > 5 \mid x\} \subseteq \{x \cdot x > 4 \mid x\}$ can be translated into a Why3 language using our definition. In this simple example we check if one set is a subset of another.

In summary the early development stages showed that with some provers the machine quickly runs out of memory, thus axiomatization process of WHY3 functions was highly constrained by state space. Whole operators axiomatization process was

done in similar manner to [MMFA12] where majority Event-B set operators, where defined in a form of membership.

```
goal g0:
(subset
  (bsetc  (\ z : (int, int) . (let (id_x, id_x1) = z in (id_x + id_x1) >
   5))
     (\ z : (int, int) . (let (id_x, id_x1) = z in id_x)))
       (bsetc (\id_x : int .(id_x > 4)) (\id_x : int .id_x)))
```

Listing 4.8: Lambda abstraction example translation

In order not to expand state space dramatically some generalization of lemmas is necessary or otherwise as previously mentioned some state-of-the-art hypothesis filtering. In Section 4.4 we provide more information on how such lemmas included into Event-B theory.

4.3 Generic Lemmas

Building a good model is necessarily a trial and error process; one often has to start from a scratch or do considerable re-factoring to produce an adequate model. This, obviously, necessitates redoing proofs and makes time spent proving dead-end efforts seem pointlessly wasted. Hence, proof-shy engineers too often do not make a good use of formal specification stage as they tend to hold on to the very first, often incoherent design.

There is a number of circumstances when existing interactive proofs become invalidated and a new version of an undischarged proof obligation appears. On rare occasions a model or its sizeable part are changed significantly so that there is no or little connection between old and new proof obligations. Far more common are incremental changes that alter the goal, set of hypotheses, identifier names or types. During the re-factoring of a refinement tree it is very common to lose a large proportion of manual proofs.

While there is a potential to improve the way Rodin Platform handles interactive proofs, the fragility of such proofs has mainly to do with their nature. Unlike more traditional theorems and lemmas found in maths textbooks, model proof obligations have no meaning outside of the very narrow model context. And since Event-B relies on syntactic proof rules for invariant and refinement checks, even fairly superficial syntactic changes would result in new proof obligations which are, in fact, if not logically equivalent are often quite similar to the deleted ones.

Even in the case of a significant model change, it is, in our experience, likely that proof obligations similar to those requiring an interactive proof re-appear. In addition, there is a large number of essentially identical interactive proofs re-appearing in different projects due to specific weaknesses in the underlying automatic provers. We hypothesise that over time a modeller develops a modelling style characterised by some fairly stable patterns in the usage of the mathematical language and specification constructs. And this leads to stable patterns in the generated proof obligations. For instance, some modellers prefer separate definitions of the direct and converse forms of a relation whilst others construct the converse form on the fly.

The key to our approach is understanding what 'similar' means in the relation to some two proof obligations. One interpretation is that similar conditions can be discharged by the same proof scripts. To make it practical, this has to be relaxed with some form of a proof script template [FW14]. The interpretation we take in this work is that two proof obligations are similar if they both can be discharged by adding same schematic lemma to the set of their hypotheses. This definition is rather intricately linked with the capabilities of underlying automated provers: adding a tautology (a proven lemma) to hypotheses does not change a conjecture but it might help to guide an automated prover to successful proof completion.

It is our experience that the existing Rodin automatic provers do not benefit from adding a schematic lemma (with instantiated type variables, to make it first order) to hypotheses and they still need to be instantiated manually by an engineer to have any effect. However, in the case of the Why3 plug-in, with which this approach has a close integration, it is different: a fitting schematic lemma in hypotheses makes proof nearly instantaneous.

Since every modelling project is likely to have a fairly distinctive usage of data types and mathematical constructs, we might also expect a distinctive set of traits in supporting lemmas. We hypothesise that such distinctness is pronounced and dictated by the modeller's experience and background as well as the model subject domain. We have also observed that the style of informal requirements - structured text, hierarchical diagram, structural diagram - has an impact on a modelling style.

There are situations when the only viable way to complete a proof is by providing a proof hint. One such case - refinement of event parameters - is adequately addressed at the modelling notation level where a user is requested to provide a witness as a part of a specification. There are proposals to generalise this, for the majority of situations, and define hints at the model level [Hoa12].

4.3.1 Schematic lemmas

We want to change the way proofs are done, at least in an industrial setting. In place of an interactive proof - something that is inherently a one-off effort in Event-B - we want to invite modellers to write and prove a non-model specific condition called a schematic lemma that would, once added to hypothesis set, discharge an open proof obligation. Such a lemma may not refer to any model variables or types and is, in essence, a property supporting the definition of the Event-B mathematical language. If such a lemma cannot be found or seems to be to difficult to prove, the model must be changed.

A schematic lemma is a tangible and persistent outcome of any modelling effort, even an abortive one. Being generic, a schematic lemma is likely to be useful in a next iteration and, as we might hope, there is a point when all relevant lemmas are collected and modelling, in a given domain and for a given engineer, is nearly completely free of interactive proofs.

A schematic lemma considered on its own is of a little use. But if a proof obligation can be proven by adding a schematic lemma, then the construction of a schematic lemma in itself a proof process. As a simple illustration, consider the following (trivial) conjecture:

$$
\begin{array}{l}
\mathit{library} \in \mathrm{BOOKS} \to \mathbb{N} \\
b \in \mathrm{BOOKS} \land c \in \mathbb{N} \\
\ldots \\
\vdash \\
\mathit{library} \lhd\!\!+ \{b \mapsto c\} \in \mathrm{BOOKS} \to \mathbb{N}
\end{array}
$$

And suppose there were no automated prover capable of discharge it. It is clear that the crux of the statement is in the interaction of functional override, totality and functionality. The above can be rewritten as

$$
\begin{array}{l}
f \in A \to B \\
\vdash \\
\forall x, y \cdot x \in A \land y \in B \Rightarrow f \lhd\!\!+ \{x \mapsto y\} \in A \to B
\end{array}
$$

Since the Event-B mathematical language does not have type variables such a condition may only be defined either for specific A's and B's, or, in a slightly altered form, using the Theory plug-in [BM13]. But to discharge the original proof obligation one still needs to find this lemma and instantiates it. It is a tedious and error-prone process for a human but a fairly trivial task for a certain kind of automated provers.

The example above is quite generic in the sense it is potentially useful for in many other contexts. At times a schematic lemma need to be fairly concrete (see examples in Section 5). It is also easier to write a lemma that narrowly targets a proof obligation. This distinction between 'general' and 'specific' is, at the moment, completely subjective and relies on the modeller's intuition. To reflect the fact that a more general lemma is more likely to be reused, schematic lemmas are classified into three visibility classes: machine (single model), project (collection of models) and global. A machine-level lemma will be considered for a proof obligation of the machine with which the lemma is associated; similarly, for the project-level attachment. A global schematic lemma becomes a part of the Event-B mathematical language definition for the Why3 plug-in.

Just as model construction is often an iterative process, we have discovered during our experiments that finding a good schematic lemma may require several attempts. A common scenario is that an existing lemma may be relaxed so that while it is still strong enough to discharge conditions that were dependent on it, it can also discharge some new ones. For instance, we have seen several cases where a fairly narrow and detailed lemma would gradually slim down to a simple (and much more valuable) statement about distributivity of certain operators. It does require at times a considerable effort to come up with an abstract and minimal covering condition but the result is rewarding and reusable across projects.

4.3.2 Schematic lemma plug-in

We have built a prototype implementation of the schematic lemma mechanism as a plug-in to Rodin Platform. It integrates into the prover perspective and offers an alternative way to conduct an interactive proof either at a root node level or indeed for any open sub-branch of a proof obligation. At the moment, the notation employed is the native notation of Why3 but the first release will support entering a schematic lemma in the Event-B mathematical notation.

There are three main parts to the definition of a schematic lemma: identifiers, hypotheses and the goal. Identifier definition may use either one of the two built-in types (boolean and integer) or a fresh type variable (i.e., $type0$ in Fig. 4.2). Hypotheses are defined by a list of predicates (while logically order should not matter, in practice it does and it is advantageous to have more constricting hypotheses first);

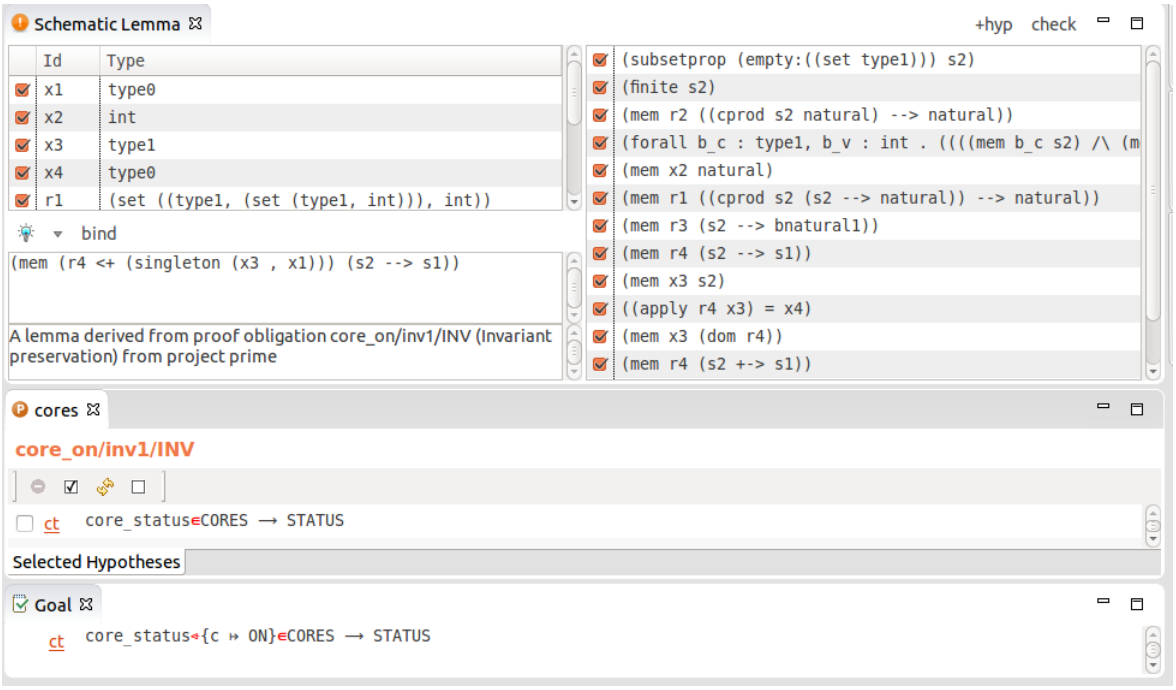

Fig. 4.2: Schematic lemma prover interface.

these predicates may not mention any model variables but can refer to the identifiers defined in the lemma. And the goal is a predicate over the lemma identifiers.

Instead of working with the built-in interactive prover, a modeller attempts to construct a provable schematic lemma that would discharge the current proof obligation.

The plug-in automatically constructs the first attempt at a schematic lemma through a simple syntactic transformation of a context proof obligation. All the identifiers occurring in either hypotheses or goal of the proof obligation are mapped into schematic lemma identifiers and then this mapping is used to translate hypotheses and the goal.

From this starting point it is up to the modeller to construct a promising lemma. A prepared lemma is *committed* where the Why3 plug-in is used to prove that the lemma holds, and also that adding it to the proof obligation in context discharges the proof obligation. If either fails, a user gets an indication of what has happened and it is not until both generic and concrete proofs are carried out that the schematic lemma may be used in the local library and assigned a binding level (machine, project or global). In the case of a success, the current open goal is closed.

To aid in the construction of a schematic lemma, the plug-in provides some simple productivity mechanisms. A hypotheses can be deselected without removing it to check whether both the lemma goal and the context proof obligation are still

provable. An identifier may also be deselected and this automatically deselects all the hypotheses mentioning the identifier. It will take more experiments to arrive at methodological guidelines on constructing lemmas.

4.4 Hypothesis and Lemma Filtering

The initial experiments have shown that a minimal axiomatisation support is not sufficient to discharge a sizeable proportion of proof obligations. Provable lemmas were added to assist with specific cases but then it become clear that a large number of support conditions slow down or even preclude a proof. On top of that, the auto tactic language of Rodin offers a very crude hypotheses selection mechanism that for larger models tends to include tens if not hundreds of irrelevant statements. It was thus deemed essential to attempt to filter out unnecessary axiomatisation definitions, Why3 support lemmas and proof obligation hypotheses.

The Rodin mechanism for hypotheses filtering is based on matching conditions with common free identifiers. To complement this mechanism we do filtering on the structure of a formula. It is also a natural choice since support lemmas do not have any free identifiers.

Directly comparing some two formulae is expensive: a straightforward algorithm (tree matching) is quadratic unless memory is not an issue. We use a computationally cheap proxy measure known as the Jaccard similarity which, as the first approximation, is defined as:

$$JS(P, Q) = \mathrm{card}(P \cap Q) / \mathrm{card}(P \cup Q) \tag{4.1}$$

The key is in computing the number of overall and common elements and, in fact, defining what an "element" means for a formula. One immediate issue is that P and Q are sets and a formula, at a syntactic level, is a tree.

One common way to match some two sequences (e.g., bits of text) using the Jaccard similarity is to use *shingles* of elements to attempt to capture some part of the ordering information. A shingle is a tuple preserving order of original elements but seen as an atomic element. Thus sequence $[a, b, c, d]$ could be characterised by two 3-shingles $P = \{[a, b, c], [b, c, d]\}$ (here $[b, c, d]$ is but a structured name) and matching based on these shingles would correctly show that $[a, b, c, d]$ is much closer to $[a, b, c, d, e]$ than to $[d, c, b, a]$. Trees are slightly more challenging.

On one hand, a tree may be seen (but not defined uniquely) as a set of paths from a root to leaves and we could just do matching on a set of sequences and aggregate the result. This is not completely satisfactory as tree structure is not accounted for. So we add another characterisation of tree as a set of sequences of the form $[p, c_1, \ldots, c_2]$ where p is a parent element and $c_1, \ldots, c_2$ are children. This immediately gives a set of n-shingles that might need to be converted into shorter m-shingles to make things practical.

As an example, consider the following expression $a * (b + c/d) + e * (f - d * 2)$. We are not interested in identifiers and literals so we remove them to obtain tree $+(*(+/))(*(-*))$ which has the following 3-shingles based on paths, $[*, +, /], [+, *, +], [+, *, -], [*, -, *]$, and only 1 3-shingle, $[+, *, *]$, based on the structure. The shingles are quite cheap to compute (linear to formula size) and match (fixed cost if we disregard low weight shingles, see below). Let $\mathrm{sd}(P)$ and $\mathrm{sw}(P)$ be set of depth and structure shingles of formula P. Then the similarity between some P and Q is computed as:

$$\begin{aligned} s(P,Q) &= \textstyle\sum_{i \in I_1} \mathrm{w_d}(i) + c \sum_{i \in I_2} \mathrm{w_w}(i) \\ I_1 &= \mathrm{sd}(P) \cap \mathrm{sd}(Q), I_2 = \mathrm{sw}(P) \cap \mathrm{sw}(Q) \end{aligned} \tag{4.2}$$

where $w_*(i) = \mathrm{cnt}(i)^{-1}$ and $\mathrm{cnt}(i)$ is number of times i occurs in all hypotheses and support lemmas. Very common shingles contribute little to the similarity assessment and may be disregarded so that there is some k such that $\mathrm{card}(I_1) < k, \mathrm{card}(I_2) < k$.

5 Verification Tool Evaluation

In this chapter we discuss the experience of applying the proposed technique to prove several pre-existing models. We also discuss perceived advantages and disadvantages of doing proofs with our technique.

In the following subsections we start by addressing the importance of the automatic part of the verification process by providing statistics on recent experiment results. Then we demonstrate an example of how the schematic lemma method was used to discharge verification goals and how lemmas propagate within a model.

5.1 Case Study

We approached the experiment in a more or less blind style where a model itself was not analysed in any detail and we were generally concerned only with the specifics of a proof obligation - its goal and hypothesis, - in an attempt to deduce a schematic lemma strong enough to discharge the condition.

As case study we consider five models, some of them fairly well known to the Event-B community. They are not very large, but still have a reasonable number of proof obligations and make a good use of refinement and Event-B modelling notation. Our intention was to take models from different domains constructed by different people to see how the technique performs in different settings.

The core of the experiment was to apply the Why3 plug-in to several diverse models and compare results with the existing proof infrastructure including the Why3 plug-in equipped with schematic lemmas. The Rodin Platform provides facility to define automatic tactics, combining certain rewrite rule and automatic provers, and apply them to redo all the proofs of a project.

For this experiment, we have defined four such tactics and compared their performance. We have made every attempt to make the best use of the available tools such as Atelier-B ML prover, built-in PP and nPP provers, and, of course, the SMT plug-in that relies on some of the same back-end SMT provers.

5.1.1 Automatic proving

In this section we present results obtained in applying automatic side of the tool. Particularly, we were interested in seeing how tool performs compared to the SMT plug-in which uses similar back-end provers. The results obtained could provide some important feedback to further improve axiomatisation of set-theoretic operators.

Table 5.1 summarises the results of our experiment. We use two tactics that are commonly available to Rodin users. Tactic[1] applies a number of rewrite rules and then tries nPP, PP and ML provers; Tactic[2] does the same with addition of the SMT plug-in. The Why3 tactic is similar to Tactic[1] but with Why3 plug-in as the sole automatic prover. This tactic does not use any schematic lemmas and relies solely on the basic axiomatisation library defining various Event-B operators. In the last column, the Why3 plug-in is able to locate and include suitable schematic lemmas. This is a completely automatic process: one can define a number of schematic lemmas (when doing interactive proofs), then purge all the proofs and the lemmas will be picked up automatically when relevant. The last number (+x) is the number of used schematic lemmas.

Model	Proof obligations	open, Tactic[1]	open, Tactic[2]	open, Why3	open, Why3 (+ SL)
Order/Supply Communication[Fur09]	276	24	4	8	4 (+2)
Fisher's Algorithm[IB15]	82	16	4	1	0 (+1)
Train Control System [Abr10] (Chapter 17)	133	36	5	32	32 (+0)
B2B Communication prot.[Hoa09]	498	63	25	20	8 (+5)
Automated Teller Machine[SBS09]	962	77	28	1	0 (+1)
Total	1951	216	66	62	44

Tab. 5.1: Comparative performance of four proof tactics

The first column is the overall number of generated proof obligations while the following four columns give the number of proof obligations remaining open (undischarged) after applying from scratch (that is, purging any previous proofs) a certain proof tactic. The final column gives in brackets the number of schematic lemmas used in the model (but not necessarily defined specifically for the model).

The analysis of the automatic part of verification provided us with some interesting results regarding preferred style of the goal, the importance of simplification rules and prover performance.

First of all, it was noticed how a simple automatic simplification rule like Conjunctive Goal which splits a goal into two sub-goals can dramatically increase the number of successfully discharged proof obligations. The reason behind it is that provers are very different from each other, not only in terms of the language they were written, but also style and type of the goal they prefer. It was noticed - that in many

instances, especially proving well-defined type of goals a specific prover would only satisfy one side of the conjunctive goal, whereas different prover another, thus only a combination of provers and automatic simplification rules would discharge it.

Furthermore, it was observed that our tool does not particularly like goals with equality and does much better if the goal is simplified using set equality rewrite rule. Integrating few of those simplification rules into the automatic prover tactic can significantly increase the number of proof obligations discharged.

With one of the models (Train Control System) the Why3 plug-in showed a lacklustre performance compared to the the SMT plug-in but we also found it hard to come up with any useful schematic lemmas. Two of the remaining models were not proven completely as we have found it quite hard to read large proof obligations and deduce what is really happening there. It should become, we hope, easier for a modeller who has a ready intuition as to what is the underlying meaning of a given proof obligation.

5.2 Schematic Lemma: Seller B2B Communication Protocol

As we have stated previously, it has been one of the goals of this research to establish to what degree schematic lemmas are reusable at least within the same project. Clearly, it would not make any sense to write a dedicated lemma for each open proof obligation.

In this experiment, we address the problem of proof re-usability by shifting the focus from proving a single verification condition to validating remaining undischarged proof obligations of the model. We use a publicly available model Buyer/Seller B2B Communication protocol [Hoa09]. In our view, it is a fairly typical example of a model not constructed solely for illustration purposes, i.e. there is some scale and purpose to it.

A Buyer/Seller B2B Communication protocol model has 11 refinement steps and 498 verification conditions. Combining all the default tactics with all the available automatic provers and the SMT plug-in results in 25 undischarged verification conditions (63 without the SMT plug-in). Our standard routine based on the Why3 plug-in consists in first applying the plug-in without any schematic lemmas with increasingly longer timeouts and only afterwards reviewing remaining conditions for the purpose of writing schematic lemmas.

Model	open, Why3	open, + L_1	open, + L_2	open, + L_3	open, + L_4	open, + L_5
B2B Communication prot.	20	16	14	12	10	8

Tab. 5.2: The dynamics of proving the B2B Communication protocol model using the schematic lemma technique. The numbers show how each next lemma (L_1, L_2, ...) affects the overall number of open proof obligations.

For this specific experiment, we used an incremental timeout tactic with three theorem provers: Z3, EProver and Alt-Ergo. The initial timeout was set to 5s then to 15s and finally 45s which roughly the point when provers start to run out of memory. The vast majority of conditions were proven under 5s, only few more between 5 and 15s, and no new conditions were proven with the 45s timeout. The Why3 plug-in on its own has discharged a significant part of the obligations: only 4.6 per cent of the 498 open verification conditions were not automatically proven which is better than the SMT plug-in.

One immediately satisfying result that the schematic lemmas defined for two other models (Order/Supply Communication and Fisher's Algorithm) - completely unrelated in terms of domain and provenance - discharged ten proof obligations of the B2B model. After that, we have added further five schematic lemmas each discharging between 2 and four proof obligations. Table 2 shows the proof progress taking the model from 20 POs to 8 via five schematic lemmas. The remaining 8 could not be easily done with this approach. We have not arrived at a definite conclusion of whether there is a sizable class of proof obligations for which one cannot construct meaningful lemmas or if it is just the case of unfamiliarity with the model making writing schematic lemmas inordinately difficult.

The fundamental idea behind this proving style is to virtually break down a statement into pieces and consider what basic properties that could be missing. For instance, the following trivial condition has discharged a large of seemingly unrelated proof obligations in several models.

```
lemma lemma_natural_increment:
  forall x, n : int.
    mem x bnatural1 ∧ n ≥ 0 →
      mem (x + n) bnatural1
```

Listing 5.1: Schematic Lemma: natural increment

A fairly common tactic in the schematic lemma approach, when not familiar with the model, and the condition appears to be true, is to try and identify the few key hypothesis and come up with a lemma that would bridge them to the goal. Although

it sounds fairly trivial, proof obligations may contain tens if not hundreds hypotheses so just visually spotting the right few one might be tricky. A heuristic approach we discussed in Section 4.4 to automatically filter and rank schematic lemma hypotheses could help with the process.

As an illustration of the finiteness consider the following simple example:

$$\begin{array}{l} \mathrm{finite}(B\_2\_S\_proposal) \\ B\_2\_S\_counter\_proposal \in B\_2\_S\_proposal \twoheadrightarrow \mathrm{dom}(B\_2\_S\_rejection) \\ \vdash \\ \mathrm{finite}(B\_2\_S\_counter\_proposal) \end{array}$$

It is not hard to prove it by hand, however, it is tedious to do it over an over again. So we added the following schematic lemma and all such similar cases are now instantly discharged.

```
lemma lemma_finite_image_function:
  forall f : rel 'a 'b, s : set 'a, t : set 'b, z : set 'b.
    mem f (s +-> t) ∧ finite s →
      (finite (image (inverse f) z))
```

Listing 5.2: Schematic Lemma: image of a finite function

5.2.1 Nesting lemmas

There are situations where a suitable schematic lemma which we believed to be correct, and which as well discharged the context proof obligation could not be proven by the Why3 plug-in. Initially, this was a puzzling scenario as one would not want to comprise on the form of a schematic lemma. A possible back-door solution is to add (in a safe way, with a proof) a lemma to the Why3 library of Event-B axiomatisation and include the lemma in every single proof obligation. However, we knew from the earlier experiments with the Why3 plug-in that a large number of supporting lemmas may overwhelm provers and then, in an extreme, pretty much nothing is provable.

The solution is to allow a modeller to construct chains of lemmas of which only the last one is used in the capacity of a schematic lemma and the rest help to prove it. With extra support lemmas one should be able to handle pretty much any case of

forward or backward proof. These additional lemmas are visible in the context and saved with the schematic lemma so that one is able to redo all the proofs strictly on the basis of Why3 axiomatisation library.

One example where we discovered a need for nesting lemmas is a relatively common case of proving that an overridden restricted relation is a member of a function. The effect of overriding $f \lhd\!\!+ \{x \mapsto y\}$ is replacing mapping $\{x \mapsto f(x)\}$ with $\{x \mapsto y\}$ in f. In example below, function $database$ is overridden by a singleton pair and one needs to check whether it remains a total function.

$$\cdots \vdash database \lhd\!\!+ \{ai \mapsto a\} \in Attr\_id \rightarrow Attrs$$

After unsuccessful attempts to prove it automatically, we used a schematic lemma technique to discharge it. Firstly, we added schematic lemma shown below.

```
lemma lemma_total_overriding:
  forall f:rel 'a 'b, s:set 'a, t:set 'b, x: 'a, y : 'b.
    mem f (s --> t) ∧ mem x s ∧ mem y t →
      mem (f <+ singleton (x, y)) (s --> t)
```

Listing 5.3: Schematic Lemma: total function overriding

It seemed to be a promising start as the original proof obligation was now discharged by Alt-Ergo (among others) in just 0.03s. Yet the lemma itself could not be proven. We discovered two new lemmas that should be added in the context of the schematic lemma and are enough to discharge it. They state some simple properties about domain overriding, and the functionality of an overridden function.

```
lemma lemma_total_overriding_help0:
  forall f : rel 'a 'b, x : 'a, y : 'b.
    subset  (dom f) (dom (f <+ (singleton (x, y))))
```

Listing 5.4: Nesting lemma for total overriding

```
lemma lemma_total_overriding_help1:
  forall f:rel 'a 'b, s:set 'a, t:set 'b, x: 'a, y : 'b.
    mem f (s --> t) ∧ mem x s ∧ mem y t →
      mem (f <+ singleton (x, y)) (s +-> t)
```

Listing 5.5: Nesting lemma for total overriding

Both statements were proven with the Alt-Ergo prover times were 1.74s and 1.08s. It is important to note that these lemmas only appear in the context of proving `lemma_total_overriding`.

5.3 Discussion and Summary

Completely automating a verification process is a largely debatable idea and a grand challenge for automated reasoning community. Nonetheless, we were keen to experiment with a handful of models and our tool which exploits modern state-of-the-art theorem provers and identify on how far are we from the ultimate objective.

The models we have chosen for the case study are not particularly large. We have on purpose avoided taking some of the large industry-constructed as they have unusually high proportion of interactive proofs and may argue that Event-B abstraction mechanisms were not used to full extent to manage complexity and reduce the proof workload. In the longer term, however, we would want to tailor our technique to the needs of an industrial user. We believe, and this is supported by our experiments, that with a carefully lemma library and a domain-specific modelling guidance document, an industrial user will be able to construct large and useful models without doing a single interactive proof. Failed proof obligations will still be reported, in slightly different style from now, to inform a modeller what is wrong and how it can be fixed. Any proof obligation remaining undischarged after throwing at it all possible automatic provers will be treated as a modelling error irrespective of whether the condition can be potentially proven or not. A similar mindset of restricting the usage of modelling notation in order to gain productivity has been used with some great success for the Classical B refinement process [BM99].

The schematic lemma technique has the potential to significantly alter the way models are proved while proof persistence encourages frequent and deep model re-factoring. We also hypothesise that at a certain stage accumulated schematic lemma make automatic proof support so complete that interactive proofs are no longer necessary and an undischarged proof is treated as failed and must be dealt with at a model level. On the whole we were pleased to find that such diverse models still share a lot of schematic lemmas and it supports our conjecture that it is worthwhile to build lemma library. We do not have enough to show that this process definitely leads to a saturation point but we did observe that each subsequent model we tackled was a little bit easier since lemmas are reused.

In general it is not easy to define the notion of a useful lemma and how one might look in our context. Simply put a good schematic lemma should help to discharge a

proof obligation. Although, in our view a good lemma should address the weakness in axiomatisation.

The following lemmas shown in Listings 5.6 - 5.7 were derived to prove Buyer/Seller B2B Communication protocol model. In total these two lemmas discharged six verification conditions of the model. If we were to analyse them it would not necessarily be easy to deduce what weakness in our axiomatisation of the Event-B method they address.

```
lemma lemma_singleton_range_subtraction_function:
  forall f : set (int, int), x : int.
    mem f (natural +-> bnatural1) ∧ mem x (dom f) →
      mem ((f <+ singleton (x, (apply f x) - 1)) |>> (singleton 0))
    (natural +-> bnatural1)
```

Listing 5.6: Schematic Lemma: range subtraction of a function

```
lemma lemma_cprod_overriding:
  forall f : rel 'a 'b, x : 'a, y : 'b, s : set 'a, t : set 'b.
    mem f (s +-> t) ∧ mem x s ∧ mem y t →
      mem (f <+ (singleton (x, y))) (pow (cprod s t))
```

Listing 5.7: Schematic Lemma: relation overriding

In essence our ultimate goal to have a large set of lemmas which could lead to a saturation point where no more lemmas are needed would benefit more if one would spend more time on deducing a more optimal schematic lemma. Clearly in the industrial setting time is too precious, but perhaps in the academic community this would be a desirable approach.

One problem we foresee is the accumulation of lemmas of which majority are irrelevant outside of some narrow context. Putting all lemmas as axioms in a proof context make proof progressively more difficult with each new lemma. It could conceivably reach a point where nothing may be proven due to the sheer number of supporting properties. Some form of filtering is thus paramount. From the outset, we make set of supporting conditions bound to a specific application and group of developers. To filter out irrelevant lemmas within such context we are looking into state-of-the-art in hypothesis filtering. At the very basic level, we start with filtering by user-defined template expressions where a lemma is included only if a goal or a hypothesis predicate matches a given template.

Generally shared feature among variety of the models is repetitive pattern and groups of undischarged proof obligations. As previously discussed, a huge disadvantage

of proving a model interactively is the proofs transience. We addressed this issue and proposed schematic lemma approach which is used in this experiment to verify the model. Although, the remaining verification conditions of a Buyer/Seller B2B Communication protocol model had different types, variables names, it was not hard to notice that many POs shared exact or similar structure. Therefore, challenge of this exercise was to come up with generic and valid lemmas which would not only prove a single goal, but could be also reusable in different context.

The idea of generalisation for the purpose of proof reuse has been explored in different settings. Perhaps the most well-known example to aspire to is the tactic or meta-proof language supported by general purpose interactive theorem provers such as Isabelle [NWP02]. It is far more flexible and powerful technique but also requires a different level of expertise from a user. A much simpler technique is having a customisable set of rewrite or simplification rules. In principle, this is offered to some extent by the Theory plug-in; the Atelier-B interactive prover allows a modeller to define custom rewrite rules although this is can be extremely unsafe. Reusable theory components with embedded lemmas, tautologies and rewrite rules are widely used in many verification tools from Maude to ACL2 and also recently available, thanks to the Theory plug-in, in Event-B. Schematic lemmas are far less topical than such theory components but then their inclusion is triggered automatically via syntactic matching rather than through direct instructions from a user.

In this work we have tried to weave the process of constructing generalised proofs into the very process of model construction and address two long standing challenges of model-based design: turning proofs into tangible artifacts that can survive deep model re-factoring, and making interactive proof on organic part of model construction rather than an unfortunate side activity.

6 Conclusion and Future Work

This section summarises the research presented in this thesis. It provides a description of the problem in question and the approach used to address it. Furthermore, it discusses the results obtained during the evaluation of the verification tool and suggests possible tool improvements as future work directions.

6.1 Conclusions

The applicability of formal methods in the industrial setting is closely tied with automating the verification process. In theory, to completely automate the process of proving is impossible, because of theorem proving being an undecidable problem. However, in a practical sense with the state of the art techniques we can dramatically improve the prospects for automation at least for a subset of problems in the industry.

In this research we have tried to address the long standing challenge of interactive proofs. There is a popular trade-off phrase - *quality over quantity* which is well suited for many problems and situations. In our case, considering the public accessibility to a number state of the art theorem provers and cheap computational power offered by the cloud technology we decided to exploit both - quality and quantity.

In the first place we attempted to reduce the number of interactive proofs one needs to complete. We proposed an adaptable and scalable verification architecture which effectively with adjusted translator could support a variety of formal specification languages. The suggested architecture could function either in a local mode or remotely as a cloud service. In the study we decided to implement our verification tool to support verification of proofs obligations of a popular specification language, Event-B.

The verification tool to the Rodin Platform was realised to map between the Event-B mathematical language and Why3 theory input notation. To begin with, we presented challenges concerning the syntactic side of the translator. Although, a syntactic translation was a seemingly trivial implementation task, it was a trial and error process, as some syntactic Why3 features were unknown or a more advanced functionality had to be used, for example - lambda abstraction. Still, the translation

of the carrier and enumerated sets and partition operator required the most effort effort due to a different type treatment in Why3.

Yet a significant effort was needed to formalise a Event-B language in Why3 theory. In developing theory of a complex language like Event-B we needed to consider and ensure following aspects: soundness, theory structure and axiomatic sufficiency. The greatest risk is an inconsistent theory from which contradicting proof obligations can be discharged. In our approach we provide just a minimal number of axioms which are sufficient to prove a number of proofs obligations, yet still we believe are sound. We accept that the axiomatisation of such a language will be an ever open problem, in particular if one wants to include extra axioms to improve verification. Because of that reason, we provide additional operator support theories where the user may contribute additional axiomatic properties. The supplementary axioms may be included manually or with an integrated syntactic filtering mechanism (see Section 4.3).

Section 5 presented results and key findings of the automatic verification part of the tool. In the study we attempted to compare our tool performance against established and available Rodin provers such as Atelier-B ML, nPP and PP, and also SMT plug-in that relies on similar back-end SMT solvers. Despite a lackluster performance on the Train Control System model Why3 verification tactic in overall performed significantly better than a combination of nPP, PP and ML provers and just a slightly better than tactic which also included the SMT plug-in. In the same section we discussed the effect of simple rewrite rules had on proving success.

A particularly satisfying finding which could support the need of quantitative proving methods, was the situation when a verification condition could only be discharged with a *Conjunctive Goal* simplification rule and two different provers, because a single prover would only prove one side of the goal. In the study we discovered a few other simplification rules which significantly improved results of our tool. For example, we noticed that our tool does not particularly like goals with equality and does perform much better if the goal is simplified using the set equality rewrite rule. In general tuning Rodin simplification rules to the Why3 tactic can significantly increase the number of proof obligations discharged, however, we still need further analysis in order to deduce suitable simplification rules.

Furthermore, we realised that with a developed verification architecture we could address two long standing challenges of model-based design: turning proofs into tangible artifacts that can survive deep model refactoring, and making interactive proof on organic part of model construction rather than an unfortunate side activity. In the thesis we proposed an approach for doing interactive proofs. Instead of doing an interactive proof - something that is an inherently one-off effort - we invite users

to write and proof a schematic lemma that is strong enough to discharge an open proof obligation. Such a lemma may not refer to any model variables or types and is, in essence, a property supporting axiomatization of the modelling language.

In particular, we emphasized that such lemmata should not only be *generic* in the sense of a model types and variables, but also it could be used in wider context. In essence we defined a useful lemma as a lemma - which addresses a weakness of our modelling language axiomatisation rather than targets single proof obligation. Undoubtedly deducing such axiomatic property is a challenging exercise as we demonstrated in the evaluation section (Section 5 - page 46).

To aid in the construction of a schematic lemma, we provided some simple productivity mechanisms, discussed in Section 4.3.2. We have built a prototype implementation of the schematic lemma mechanism as a plug-in to Rodin Platform. The plug-in automatically constructs the first attempt at a schematic lemma through a simple syntactic transformation of a context proof obligation. Furthermore, we discussed possible approaches in deriving a useful lemmata, yet still, it is up to the modeller to construct a sensible lemma.

We evaluated this technique on a number of pre-existing models and we believe it demonstrated a good potential. An immediately satisfying result of this exercise was that several schematic lemmas we deduced in the previous projects *propagated* and helped to discharge ten proof obligations of the Order/Supply Communication B2B model. Even though, we do not have enough data to demonstrate yet, but we believe that at a certain saturation point accumulated schematic lemmas with an accompanied filtering mechanism could significantly increase automated theorem proving performance.

In our work we also argued that there are structurally similar or even identical proof obligations re-appearing in different projects due to specific weaknesses in the underlying automatic provers. The evaluation of the schematic lemma approach demonstrated that a single schematic property can in many instances discharge few proof obligations. In particular, we demonstrated this phenomena for Order/Supply Communication B2B model in Table 5.2 where numbers show how each next lemma $(L_1, L_2, ...)$ affects the overall number of open proof obligations.

Furthermore, we discovered that there is a fine interplay between the functioning of the schematic lemmas plug-in and the Why3 plug-in filtering mechanism. The Why3 plug-in uses the shingles technique to rank and filter hypothesis and originally aimed at just filtering out irrelevant hypothesis. We had to slightly adjust matching weights as there are no common identifiers between a proof obligation and a schematic lemma so a bigger emphasise has to be made on structural patterns. In particular,

instance we discovered a seemingly trivial schematic lemma (Listing 5.1) which has discharged a number of proof obligations. However, the current implementation of filtering mechanism simply relies on syntactic analysis, therefore, there might be a scenario when a useful lemma might be filtered out.

In this thesis we presented a cloud-based verification tool which enables modeller to exploit a number of existing theorem provers to prove Event-B models. The obtained results in the study are certainly encouraging, as at least for a number of Event-B models the degree of verification automation was increased compared to existing techniques. On the other hand, the verification tool for some models (e.g. Train Model [Abr10]) performed worse, thus, further improvements (Section 6.2) are necessary. Finally we stated that a success of formal methods will largely depend on our community to produce and improve new methods and technique. We believe that this statement reflects in our work as limitation of the verification tool lies on a formal methods community to continue producing the state of the art theorem provers.

6.2 Future work

This thesis presented a research study which can be extended in a number of ways. This section contains an overview of potential directions for the verification tool improvements and future work. The current version of the plug-in is open source and is available upon request. Nonetheless, the foremost future objective is to release it with a public cloud service in the coming few months. In collecting some tens of thousands of proof obligations we could further improve our axiomatisation and potentially discover subtle translation or axiomatisation errors.

The case study we completed provides an important feedback on several verification tool improvement areas. As one extension of this work we see investigation of guidelines on schematic lemma construction to help an engineer decide when and what kind of a schematic might be used. The Why3 plug-in may optionally record all the proof attempts in a database. We would like to explore whether a form of automated data mining of failed proof obligations may be employed to automatically synthesise schematic lemma candidates.

The cloud service keeps a detailed record of each proof attempt along with (possibly obfuscated) proof obligations, supporting lemmas and translation rules. There is a fairly extensive library of Event-B models constructed over the past 15 years and these are already a source of proof obligations. Some of these come from academia and some from industry. We are now starting to put models through our prover plug-in in order to collect some tens of thousands of proof obligations. One

immediate point of interest is whether one can train a classification algorithm to make useful prediction of relative prover performance. If such a prediction can yield statistically significant results, prover call order may be optimized to minimize resource utilization while retaining or improving average proof time.

Proof scenarios like FA and SA (see Section 3.2) attempts to spawn some n prover instances. This is not workable: proof requests may arrive at an arbitrary rate and takes a considerable amount of time to shutdown and clean-up a process. With a naive strategy of a running a native process for each prover, the service crashes quickly as nodes start to run out of Unix process handles. In addition, the load-balancing mechanism of Amazon EC2 we originally used seems to perform sub-optimally and some nodes overload rather promptly. Clearly, there has to be a mechanism to delay or even reject request at the times of peak demand without having to run into the hard limits of underlying OS. A promising direction seems to be an agent-like abstraction where a fixed number of agents (corresponding to the total allocatable resources) pick and server requests within a logically shared tuple space.

Throughout this study we were also involved in work on applying formal methods to the railway domain. The SafeCap project [IR12] was started with the objective to overcome railway capacity problems without weakening safety constraints. The key outcome of the first stage of the project was a SafeCap tooling environment, allowed railway engineers to design and to formally and automatically verify safety. In paper by Stankaitis et al. [SI16] we discussed how the verification architecture we proposed could be integrated in a specific domain. An interesting future exercise is to check whether a theory axiomatisation could be tuned for a specific domain.

In the longer term, however, we would like to tailor our technique to the needs of an industrial user. We believe, and this is supported by our experiments, that with a carefully lemma library and a domain-specific modelling guidance document, the industrial user will be able to construct large and useful models without doing a single interactive proof. Failed proof obligations will still be reported, in slightly different style from now, to inform a modeller what is wrong and how it can be fixed. Any proof obligation remaining undischarged after throwing at it all possible automatic provers will be treated as a modelling error irrespective of whether the condition can be potentially proven or not. A similar mindset of restricting the usage of modelling notation in order to gain productivity has been used with some great success for the Classical B refinement process [BM99].

Bibliography

[Abr07] Jean-Raymond Abrial. „Formal Methods: Theory Becoming Practice“. In: *J. UCS* 13.5 (2007), pp. 619–628 (cit. on pp. 2, 3, 15).

[Abr10] Jean-Raymond Abrial. *Modelling in Event-B*. Cambridge University Press, 2010 (cit. on pp. 8, 11, 42, 54).

[Abr96] Jean-Raymond Abrial. *The B-Book*. Cambridge University Press, 1996 (cit. on pp. 8, 11).

[AIER14] Dejanira Araiza-Illan, Kerstin Eder, and Arthur Richards. „Formal verification of control systems' properties with theorem proving“. In: *Control (CONTROL), 2014 UKACC International Conference on*. IEEE. 2014, pp. 244–249 (cit. on p. 16).

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. „Météor: A Successful Application of B in a Large Project“. In: *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20–24, 1999 Proceedings, Volume I*. Ed. by Jeannette M. Wing, Jim Woodcock, and Jim Davies. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 369–387 (cit. on p. 10).

[BDDMOS13] Clark Barrett, Morgan Deters, Leonardo De Moura, Albert Oliveras, and Aaron Stump. „6 years of SMT-COMP“. In: *Journal of Automated Reasoning* 50.3 (2013), pp. 243–277 (cit. on p. 16).

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. „Why3: Shepherd Your Herd of Provers“. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64 (cit. on p. 16).

[BFT10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. „The SMT-LIB Standard Version 2.6“. In: (2010) (cit. on p. 16).

[BH95] Jonathan P. Bowen and Michael G. Hinchey. „Ten commandments of formal methods“. In: *Computer* 28.4 (1995), pp. 56–63 (cit. on p. 8).

[BHR84] Stephen D. Brookes, Charles A. R. Hoare, and Andrew W. Roscoe. „A Theory of Communicating Sequential Processes“. In: *J. ACM* 31.3 (June 1984), pp. 560–599 (cit. on p. 8).

[BJ78] Dines Bjørner and Cliff B. Jones, eds. *The Vienna Development Method: The Meta-Language*. London, UK, UK: Springer-Verlag, 1978 (cit. on p. 8).

[BJRT06] Michael J. Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, eds. *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*. Vol. 4157. Lecture Notes in Computer Science. Springer, 2006 (cit. on p. 13).

[BM13] Michael Butler and Issam Maamria. „Practical Theory Extension in Event-B“. In: *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*. Ed. by Zhiming Liu, Jim Woodcock, and Huibiao Zhu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 67–81 (cit. on pp. 13, 36).

[BM99] Lilian Burdy and Jean-Marc Meynadier. „Automatic refinement“. In: *Proceedings of BUGM at FM 99* (1999) (cit. on pp. 47, 55).

[Bri10] James P. Bridge. „Machine learning and automated theorem proving“. PhD thesis. University of Cambridge, 2010 (cit. on p. 16).

[BS89] R. J. R. Back and K. Sere. „Stepwise refinement of action systems“. In: *Mathematics of Program Construction: 375th Anniversary of the Groningen University International Conference Groningen, The Netherlands, June 26–30, 1989 Proceedings*. Ed. by J. L. A. van de Snepscheut. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 115–138 (cit. on p. 11).

[BS93] Jonathan Bowen and Victoria Stavridou. „Safety-critical systems, formal methods and standards“. In: *Software Engineering Journal* 8.4 (1993), pp. 189–209 (cit. on p. 2).

[CCGR00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. „NuSMV: a new symbolic model checker“. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425 (cit. on p. 9).

[CW96] Edmund M. Clarke and Jeannette M. Wing. „Formal Methods: State of the Art and Future Directions“. In: *ACM Comput. Surv.* 28.4 (Dec. 1996), pp. 626–643 (cit. on p. 8).

[DB09] Kriangsak Damchoom and Michael Butler. „Applying event and machine decomposition to a flash-based filestore in Event-B“. In: *Brazilian Symposium on Formal Methods*. Springer. 2009, pp. 134–152 (cit. on p. 11).

[DFGV12] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. „SMT solvers for Rodin“. In: *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*. Springer. 2012, pp. 194–207 (cit. on p. 14).

[DLL62] Martin Davis, George Logemann, and Donald Loveland. „A Machine Program for Theorem-proving“. In: *Commun. ACM* 5.7 (July 1962), pp. 394–397 (cit. on p. 15).

[DR14] Joeri De Ruiter. „Automated algebraic analysis of structure-preserving signature schemes.“ In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 590 (cit. on p. 17).

[DT14] Robert Dockins and Andrew Tolmach. „SUPPL: A flexible language for policies“. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2014, pp. 176–195 (cit. on p. 17).

[ERB12] Andrew Edmunds, Abdolbaghi Rezazadeh, and Michael Butler. „Formal modelling for Ada implementations: Tasking event-B“. In: *International Conference on Reliable Software Technologies*. Springer. 2012, pp. 119–132 (cit. on p. 14).

[FFM13] Alessandro Fantechi, Wan Fokkink, and Angelo Morzenti. „Some trends in formal methods applications to railway signaling“. In: *Formal methods for industrial critical systems: a survey of applications. IEEE Computer Society Press, Washington, DC* 6 (2013), p. 167183 (cit. on p. 10).

[Fit96] Melvin Fitting. *First-order Logic and Automated Theorem Proving (2Nd Ed.)* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996 (cit. on p. 15).

[Fur09] Andreas Furst. „Event-B model of the Order/Supply Chain A2A Communication“. In: (2009). Available at http://deploy-eprints.ecs.soton.ac.uk/129/ (cit. on p. 42).

[FW14] Leo Freitas and Iain Whiteside. „Proof Patterns for Formal Methods“. In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. 2014, pp. 279–295 (cit. on p. 34).

[Hal90] Anthony Hall. „Seven myths of formal methods“. In: *IEEE software* 7.5 (1990), pp. 11–19 (cit. on pp. 2, 8).

[Hal99] Anthony Hall. „Using Formal Methods to Develop an ATC Information System“. In: *Industrial-Strength Formal Methods in Practice*. Ed. by Michael G. Hinchey and Jonathan P. Bowen. London: Springer London, 1999, pp. 207–229 (cit. on p. 10).

[Har87] David Harel. „Statecharts: A Visual Formalism for Complex Systems“. In: *Sci. Comput. Program.* 8.3 (June 1987), pp. 231–274 (cit. on p. 8).

[Hax10] Anne E. Haxthausen. „An introduction to formal methods for the development of safety-critical applications“. In: *Technical University of Denmark* (2010) (cit. on pp. 8, 10).

[HK91] Iain Houston and Steve King. „CICS project report experiences and results from the use of Z in IBM“. In: *VDM'91 Formal Software Development Methods: 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, October 1991 Proceedings*. Ed. by S. Prehn and W. J. Toetenel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 588–596 (cit. on p. 10).

[Hoa09] Thai Son Hoang. „Event-B model of the Buyer / Seller B2B Communication“. In: (2009). Available at http://deploy-eprints.ecs.soton.ac.uk/128/ (cit. on pp. 42, 43).

[Hoa12] Thai Son Hoang. „Proof Hints for Event-B“. In: *CoRR* abs/1211.1172 (2012) (cit. on p. 34).

[Hol03] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. First. Addison-Wesley Professional, 2003 (cit. on p. 9).

[IB15] Alexei Iliasov and Jeremy Bryans. „Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers“. In: ed. by I. Virbitskaite V. Andrei. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. Chap. A Proof-Based Method for Modelling Timed Systems, pp. 161–176 (cit. on p. 42).

[Ili08] Alexei Iliasov. „Design Components“. PhD thesis. School of Computing Science, Newcastle University, 2008 (cit. on p. 1).

[IR12] Alexei Iliasov and Alexander Romanovsky. „SafeCap Domain Language for Reasoning About Safety and Capacity“. In: *Proceedings of the 2012 Workshop on Dependable Transportation Systems/Recent Advances in Software Dependability*. WDTS-RASD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–10 (cit. on p. 55).

[Kni02] John C. Knight. „Safety critical systems: challenges and directions“. In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. 2002, pp. 547–550 (cit. on p. 1).

[LB03] Michael Leuschel and Michael Butler. „ProB: A Model Checker for B“. In: *FME 2003: Formal Methods: International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings*. Ed. by Keijiro Araki, Stefania Gnesi, and Dino Mandrioli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 855–874 (cit. on p. 9).

[LB08] Michael Leuschel and Michael Butler. „ProB: an automated analysis toolset for the B method“. In: *International Journal on Software Tools for Technology Transfer* 10.2 (2008), pp. 185–203 (cit. on p. 14).

[LHHRO91] N. G. Leveson, M. Heimdahl, H. Hildreth, J. Reese, and R. Ortega. „Experiences Using Statecharts for a System Requirements Specification“. In: *Proceedings of the 6th International Workshop on Software Specification and Design*. IWSSD '91. Como, Italy: IEEE Computer Society Press, 1991, pp. 31–41 (cit. on p. 10).

[MB08] Leonardo De Moura and Nikolaj Bjørner. „Z3: An Efficient SMT Solver“. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer, 2008 (cit. on p. 16).

[MMFA12] David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. „Discharging proof obligations from Atelier B using multiple automated provers“. In: *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*. Springer. 2012, pp. 238–251 (cit. on pp. 27, 28, 33).

[MS11] Dominique Méry and Neeraj Kumar Singh. „Automatic Code Generation from event-B Models“. In: *Proceedings of the Second Symposium on Information and Communication Technology*. SoICT '11. Hanoi, Vietnam: ACM, 2011, pp. 179–188 (cit. on p. 14).

[MV09] Christophe Métayer and Laurent Voisin. „The Event-B mathematical language“. In: *Systerel, March* (2009) (cit. on pp. 12, 27).

[NW01] Andreas Nonnengart and Christoph Weidenbach. „Computing Small Clause Normal Forms.“ In: *Handbook of automated reasoning* 1.335-367 (2001), p. 3 (cit. on p. 15).

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. 2002 (cit. on p. 49).

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981 (cit. on p. 8).

[Rob65] John Alan Robinson. „A machine-oriented logic based on the resolution principle“. In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41 (cit. on p. 15).

[RT13] Alexander Romanovsky and Martyn Thomas. *Industrial deployment of system engineering methods*. Springer, 2013 (cit. on pp. 13, 15).

[Röd10] Jann Röder. „Relevance filters for Event-B“. In: (2010) (cit. on p. 14).

[SB06] Colin Snook and Michael Butler. „UML-B: Formal modeling and design aided by UML“. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15.1 (2006), pp. 92–122 (cit. on p. 14).

[SBS09] Mar Yah Said, Michael Butler, and Colin Snook. „Language and Tool Support for Class and State Machine Refinement in UML-B“. In: (2009). Available at http://deploy-eprints.ecs.soton.ac.uk/95/ (cit. on p. 42).

[SI16] Paulius Stankaitis and Alexei Iliasov. „Safety Verification of Heterogeneous Railway Networks“. In: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification: First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*. Ed. by Thierry Lecomte, Ralf Pinger, and Alexander Romanovsky. Cham: Springer International Publishing, 2016, pp. 150–159 (cit. on p. 55).

[Smu95] Raymond M. Smullyan. *First-order logic*. Courier Corporation, 1995 (cit. on p. 16).

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989 (cit. on p. 10).

[SS06] Geoff Sutcliffe and Christian Suttner. „The state of CASC“. In: *AI Communications* 19.1 (2006), pp. 35–48 (cit. on p. 16).

[VA14] Laurent Voisin and Jean-Raymond Abrial. „The rodin platform has turned ten“. In: *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer. 2014, pp. 1–8 (cit. on p. 13).

[Vel15] Andrius Velykis. „Capturing Proof Process“. PhD thesis. School of Computing Science, Newcastle University, 2015 (cit. on p. 3).

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. „Formal methods: Practice and experience“. In: *ACM computing surveys (CSUR)* 41.4 (2009), p. 19 (cit. on pp. 8, 10).