Invariant Discovery and Refinement Plans for Formal Modelling in Event-B

Maria Teresa Llano Rodriguez

Submitted for the degree of Doctor of Philosophy
Heriot-Watt University
School of Mathematical and Computer Sciences
December, 2012

Abstract

The continuous growth of complex systems makes the development of correct software increasingly challenging. In order to address this challenge, formal methods offer rigorous mathematical techniques to model and verify the correctness of systems. *Refinement* is one of these techniques. By allowing a developer to incrementally introduce design details, refinement provides a powerful mechanism for mastering the complexities that arise when formally modelling systems. Here the focus is on a posit-and-prove style of refinement, where a design is developed as a series of abstract models introduced via refinement steps. Each refinement step generates proof obligations which must be discharged in order to verify its correctness – typically requiring a user to understand the relationship between modelling and reasoning.

This thesis focuses on techniques to aid refinement-based formal modelling, specifically, when a user requires guidance in order to overcome a failed refinement step. An integrated approach has been followed: combining the complementary strengths of bottom-up *theory formation*, in which theories about domains are built based on basic background information; and top-down *planning*, in which meta-level reasoning is used to guide the search for correct models.

On the theory formation perspective, we developed a technique for the automatic discovery of invariants. Refinement requires the definition of properties, called *invariants*, which relate to the design. Formulating correct and meaningful invariants can be tedious and a challenging task. A heuristic approach to the automatic discovery of invariants has been developed building upon simulation, proof-failure analysis and automated theory formation. This approach exploits the close interplay between modelling and reasoning in order to provide systematic guidance in tailoring the search for invariants for a given model.

On the planning perspective, we propose a new technique called *refinement plans*. Refinement plans provide a basis for automatically generating modelling guidance when a step fails but is close to a known pattern of refinement. This technique combines both modelling and reasoning knowledge, and, contrary to traditional pattern techniques, allow the analysis of failure and partial matching. Moreover, when the guidance is only partially instantiated, and it is suitable, refinement plans provide specialised knowledge to further tailor the theory formation process in an attempt to fully instantiate the guidance.

We also report on a series of experiments undertaken in order to evaluate the approaches and on the implementation of both techniques into prototype tools. We believe the techniques presented here allow the developer to focus on design decisions rather than on analysing low-level proof failures.

To the memory of my father, Guillermo Llano, and my Grandma, Yayita.
You will always be in my heart.

Acknowledgements

First and foremost I would like to thank my supervisors Andrew Ireland, Gudmund Grov and Rob Pooley for their support and guidance. I feel extremely lucky I had Andrew as my supervisor. His expertise has made this time exciting and a continuing enjoyable learning experience. But more than a supervisor Andrew has been a role model for me. Not only as a professional but also as a person and as a parent. The dedication to his family and his endless kindness have been an example of life to me. I want to deeply thank Andrew for believing in me, for always giving me advice and for all his support, specially when I moved to Cambridge and Oxford. Also, I would like to extent my gratitude to Gudmund, for all his feedback and for making this experience very exciting with all the great ideas that never stop popping out of his mind. Finally, I would like to thank Rob for introducing me to the academic world during my MSc thesis, and for his support and guidance during the first years of my PhD.

I would also like to thank my PhD examiners, Professor Michael Leuschel and Professor Greg Michaelson, for all their comments which helped in the improvement of this thesis.

I am also very grateful to Alison Pease, for collaborating with me in the development of parts of this thesis. The time working with her was always very pleasant. I also thank Simon Colton and John Charnley, for their assistance on the work Alison and I carried out; and Jens Bendisposto for his assistance with the ProB simulator.

I thank Heriot-Watt University, EPSRC and BAE systems for funding my PhD and making the research presented in this thesis possible. In particular, I would like to thank Ben Gorry, for supporting the application of funding at BAE and for all his support during my PhD. Special thanks also go to the staff of the School of Mathematical and Computer Sciences at Heriot-Watt for their constant assistance during these years. In particular, I would like to express my gratitude to Christine McBride, who always had a smile while helping me uncountable times with expenses claims and visa documents; as well as June Maxwell and Claire Porter, for their assistance and kindness.

I would also like to thank so many friends that have been there with me during the road to completing my PhD. To my best friends, Simba and Pato, for being there always for me. To Saskia and Willem, for making the time in Edinburgh a lot more fun. To Lorenzo and Giulia, for baking those delicious desserts. To Ewen and Emma, for giving me the opportunity of listening to their beautiful music. To Amanda, for the nice lunch time we spent together in Edinburgh. To Carolina and Julian, for their unconditional friendship. To Grant and Pierre, for sharing with my husband and I very important moments in our lives, and finally, to all

my friends from Colombia, because the memories of our time together have always given me happiness.

I am specially grateful to Andrew's family; Maria, Emily, Gemma and of course Joey and Cristal, for welcoming me into their home. Special thanks goes to Gemma for the very nice cards I got every time I visited them. I also want to thank Gudmund, Ellen and Alison for receiving me in their houses. I hope one day I can return the favour to all of them.

I would like to express my deepest gratitude and love to my family. My beloved husband, Julian, my mother, Maria del Carmen, my brothers, Julio and Carlos, my nieces, Danna and Camila, my sisters in law, Marisol, Elena and Paula, and my mother in law, Cecilia. I want to thank them because without their support I would not have been able to come to the UK and achieve this goal. Mom I specially thank you for giving me so much love, for always looking after me, for everything you have done, everything you have given me, and most important, because everything good about me came from you mom. Te amo con todo mi corazon gordita. And my brothers, who have taught me by example the value of hard work, thank you for always looking after their little sister, for sharing with me so many fun memories and for giving me their unconditional and infinite love. Los quiero muchisimo hermanitos!

Julian, I cannot express in words what you mean to me. You are everything in my life, my love, my friend and my confident. None of this would have been possible without your love and support. You have helped me in so many ways, with the research during my PhD, encouraging me when I was feeling lost, making me laugh when I was stressed and just giving me so much love. Julian, you have the most beautiful heart and I am so grateful for every day I spend with you. You are the happiness in my day and the biggest inspiration in my life. I love you so much and I cannot wait to spend the rest of our lives together.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and context

The development of software has become increasingly important due to the continuous growth of complex systems across different areas in modern life. However, software often contains errors. This increases the development costs, and makes software systems vulnerable to malicious attacks, as well as fatal failures in critical applications. Guaranteeing the reliability of the software embedded in systems is therefore crucial. Traditional testing methodologies are often not enough to ensure the correctness of large and complex systems because of the size of the state space that must be explored. Moreover, testing works at the level of code where the costs of fixing are higher. Formal methods provide mathematical rigorous techniques for the modelling and verification of systems. Specifically, the application of formal methods at the design stage results in the creation of precise models that can be formally verified early during the development cycle. This not only increases the reliability and correctness of systems but also brings significant benefits in terms of development time and costs.

A recent survey by Woodcock et al. [119] reports on the current use of formal techniques in 62 industrial projects. The survey showed that formal verification is increasingly used in the development of systems across different critical and non-critical domains such as: transport, nuclear, healthcare, resource planning and automated car parking. Furthermore, the survey showed that a range of formal techniques are being used within industry, with formal specification being the most common and formal proof being the least used. The report concludes that on average there was an overall positive response towards the use of formal methodologies within industry. However, formal methods have not been widely adopted in practice. As a result, there is still a need for evolving these techniques and increasing their accessibility. This need has been identified by the Verified Software Initiative (VSI), the goal of which is:

"to establish the viability of verification as a core technology for developing reliable software" [61].

The VSI identified three key aspects in order to achieve this goal: i) to focus on formal techniques that enable the development of software systematically, including the integration of these techniques with programming languages; ii) to increase the power of tools for the automatic construction and verification of software; and iii) to evaluate such tools via the use of experiments relevant to industry.

This thesis contributes to achieving this goal by supporting refinement-based formal modelling, where a design is developed as a series of abstract models – level by level concrete details are progressively introduced via provably correct incremental steps. The techniques developed in this thesis are part of a more generic vision called *Reasoned Modelling* (REMO), introduced by Ireland et al. in [69], which combines reasoning and modelling knowledge in order to abstract away from the complexities of proofs by providing high-level modelling guidance. The long-term goal of reasoned modelling is to improve the accessibility of formal methodologies and to allow experienced users to make better use of their time by focusing on the modelling activities rather than in low-level reasoning tasks.

The techniques investigated in this thesis have been prototyped and evaluated via a series of experiments, drawing upon various developments from the literature as well as two case studies derived from the verification grand challenge [117]:

- the Mondex electronic system, a smart card purse system developed by bank NatWest[1],
- and a flash file system case study proposed by Joshi and Holzman[2] [78].

Moreover, the techniques have been explored for the Event-B formalism [3]. Event-B promotes an incremental style of formal modelling where each step of a development is underpinned by formal reasoning. In order to support the activity of refinement, this thesis has focused on two aspects: the automatic discovery of invariants and the use of planning for the generation of automatic guidance when a development fails to verify, but is close to a known pattern of refinement.

Automatic invariant discovery

Invariants play an important role when proving the consistency between the behaviour of a concrete model and the abstract model it refines. The presence of invariants in a model ensures that the properties expressed by them are not violated by subsequent refinement steps. Furthermore, invariants prevent the introduction of errors when changes are made to a

[1]The formal model of the Mondex smart card used in the experiments was developed by Butler et al.[22].
[2]The formal model of the flash file system used in the experiments was developed by Damchoom [39].

model; conversely, their absence increases the possibilities of errors being introduced into a model when the system evolves. Different types of invariants are required when modelling a system using Event-B:

- *system invariants*, which describe requirements of the system,
- *gluing invariants*, which relate the abstract state with the concrete state, and
- *technical invariants*, which act as intermediate lemmas required for their proofs to be discharged.

From a theoretical perspective these invariants are typically not very challenging. They are, however, numerous and represent a significant obstacle to increasing the accessibility of formal refinement approaches such as Event-B.

Different approaches to automatic invariant discovery have been explored. These approaches may be *dynamic*, such as the Daikon system [44] which explores whether invariant templates hold on traces that represent program executions; *constraint-based*, such as [17] in which the Alloy analyser is used to discover retrieve relations from Z specifications; or based on *term synthesis*, such as that in [90] which is used to find loop invariants by combining program analysis and proof planning. Here, a novel technique is developed based upon Automated Theory Formation (ATF) [32] and simulation. Crucially, proof failure analysis is used to constrain the search of invariants that are required to prove a refinement step.

Proof failure analysis is often used to discover invariants that are required to prove refinement steps. For instance, in the model of the Mondex electronic purse developed in [22], it is explained in detail how a few iterations of proof failure analysis are used to manually strengthen the set of invariants which are required to prove the refinement of the states of a transaction. Additionally, in [3] Abrial highlights that the productive use of proof failure not only aids understanding of the model but also assists in the discovery of missing invariants and in the strengthening of existing ones. In particular, Abrial uses proof failure analysis to obtain clues about the invariants that are required to prove strong synchronisation between an action and a reaction in a mechanical press controller system [3, Chap. 3]. Automating these kind of analyses should increase the productivity of users and improve the accessibility of formal modelling methods. Here, this manual process has been significantly automated through a prototype tool called HREMO, which identified a subset of the invariants discovered in [22] that are enough to prove the refinement step.

This approach to invariant discovery assumes that the refinement step being analysed is correct; that is, the concrete model is consistent with the abstract model. However, the process of finding a "correct" refinement typically involves exploring many incorrect models. The application of the approach is restricted to cases in which an invariant that helps discharge a current proof failure is found; otherwise, if no invariant is discovered the analysis

terminates. For such cases, a planning technique has been developed in this thesis which handles instances of incorrect models. The integration between these two approaches has also been explored and reported in this thesis.

The use of planning for refinement-based modelling

Discovering the right levels of abstraction and ensuring correctness of a design are also challenging tasks. Developers often start their developments by being too concrete or perform refinement steps that are too large. As a result requirements may not be naturally expressed, the link with informal requirements may become too large, models are often cluttered with details making it hard to understand them, debug them, or find a good proof strategy, and proof automation may be decreased. Techniques that analyse a model and provide feedback would help improve the quality of the refinement steps carried out in a development, for instance when the model fails to verify.

There are two major approaches in achieving refinement-based formal modelling. Firstly, within the *rule-based* approach, a user is restricted to a provably correct set of refinement steps – thereby ensuring the soundness of their development. An example of this style of development is found in [99]. Secondly, within the *posit-and-prove* approach, a user is free to "posit" models, but they are required to formally prove correctness of successive layers of abstraction. Examples of posit-and-prove approaches are VDM [75], B [2] and Event-B [3].

The work reported here aims to enhance the posit-and-prove approach. A number of tools and techniques exist to support refinement across a spectrum of formalisms. Often they focus on automating the refinement from a given step to a more concrete step – reducing the gap with rule-based approaches. Instances of these are: the BART tool for classical B [107]; the ZRC refinement calculus for Z [25]; and more relevant to the work presented here, Event-B based tools and techniques as described in [66, 67, 6, 60]. Contrary to those techniques, this thesis focuses on correcting a refinement step which almost matches an existing pattern, exploiting both refinement patterns and failure-analysis in order to provide alternative solutions to a broken refinement step.

More specifically, this thesis presents a technique called *refinement plans*, which automatically generates guidance for users within posit-and-prove formal modelling. This technique identifies matches between patterns of refinements and refinement steps. When a partial match is found, the failure is analysed and compared with common patterns of failures associated with the pattern. If a match is found, modelling guidance is generated which addresses the failure. The guidance may be in the form of complete solutions or partially instantiated schemas – where the guidance corresponds to a partially instantiated invariant schema. We show via a number of experiments how HREMO can be used to attempt to fully instantiate the guidance. Finally, the refinement plans mechanism has been implemented in

a prototype tool called REMO.

The research reported in this thesis was supported by EPSRC grants EP/F037058 and EP/J001058, as well as by a BAE systems studentship.

1.2 Research contributions

The research contributions of this thesis are as follows:

1. *The development of a novel approach to the automatic discovery of invariants of formal models through ATF*, which uses a set of heuristics based on proof-failure analysis to tailor the theory formation routine to the requirements of the formal methods domain. This approach effectively identifies conjectures about a model that represent candidate invariants.

2. *The development of refinement plans*, a new pattern-based technique that supports a refinement style of modelling by exploiting partial pattern matching and proof-failure analysis in order to provide high-level modelling guidance when a refinement step fails to verify but is close to a known pattern of refinement.

3. *The design and implementation of HRemo*, a prototype tool that supports automatic invariant generation for formal models, which extends HR, the leading system in the field of ATF, by automatically generating domain information from a formal model. HRemo bases the theory formation on the requirements of the input model and selects interesting conjectures that represent candidate invariants for the model.

4. *The design and implementation of REMO*, a prototype tool that supports refinement plans. REMO automatically classifies the patterns of refinement that appear in a formal development and offers automatic guidance when the input development has failures that can be associated with pattern-related failures.

5. *An experimental integration of HRemo and REMO*, which shows the potential of a framework to support refinement-style of modelling in which: i) partially instantiated guidance templates may be completed by HRemo, giving greater flexibility to refinement plans, and ii) specialised knowledge obtained from the application of refinement plans can be used to further tailor the search for invariants, increasing the probability of success by HRemo.

6. *A set of experiments that evaluate the effectiveness of the approaches*, which involve a number of developments taken from the literature as well as case studies derived from

the VSI. The set of experiments illustrates the application of the approaches in various domains as well as showing their effectiveness.

Publications

A number of publications have arisen during the development of this thesis:

- An early version of Chapter 3 appears in: Maria Teresa Llano, Andrew Ireland, Alison Pease. *Discovery of Invariants through Automated Theory Formation*. In Proceedings of the 15th International Refinement Workshop. EPTCS, 2011. [89].

- Chapters 3, 4 and 5 appear in: Maria Teresa Llano, Andrew Ireland, Alison Pease. *Discovery of Invariants through Automated Theory Formation*. In *Journal of Formal Aspects of Computing*. Springer, 2012. [88]

- Chapter 6 appears in: Gudmund Grov, Andrew Ireland, Maria Teresa LLano. *Refinement plans for informed formal design*. In *Proceedings of Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference (ABZ 2012)*. Lecture Notes in Computer Science. Springer, 2012. [55]

- Parts of chapter 7 appear in: Andrew Ireland, Gudmund Grov, Maria Teresa Llano, Michael Butler. *Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance*. In *Science of Computer Programming Journal*. 2013. [70]

1.3 Thesis structure

The rest of the thesis is organised as follows:

Chapter 2. Introduces the field of formal methods by briefly outlining different formalisms and techniques for the verification of formal specifications. Moreover, the relevant background about the Event-B formalism and the ATF machine learning technique is provided; in particular, the HR system is described.

Chapter 3. Describes the heuristic approach to invariant discovery. This shows how to use ATF and in particular HR to form theories about Event-B models and to select candidate invariants from the conjectures formed within the theories.

Chapter 4. Introduces HREMO, the prototype tool that implements the invariant discovery approach. The design of the system is described as well as some experimental results.

Chapter 5. Discusses various aspects of HREMO. First, it presents a comparative study with Daikon, a dynamic invariant generator. Second, it illustrates the generality of the approach by showing how it can be mapped to a different formalism: the Z notation. Finally, it discusses further benefits of the invariant discovery approach and the potential to extend the technique to manage incorrect models.

Chapter 6. Describes the refinement plans approach. It introduces a refinement pattern classification based on common patterns of refinement found in the literature and through manual inspection of various Event-B developments. Two refinement plans associated with patterns specified in the classification are developed and examples illustrating their applications are also provided.

Chapter 7. Introduces REMO, the prototype system that implements refinement plans. The design of the tool, as well as experimental results in the identification of patterns and the application of the refinement plans are described.

Chapter 8. Discusses strengths and limitations of the approaches introduced in this thesis. Moreover, it outlines future directions and conclusions.

Chapter 2

Background

In this chapter a brief description of the field of formal methods and ATF is provided. In particular, refinement, invariant discovery and automated reasoning techniques are outlined. Moreover, an overview of Event-B is provided, the formalism for which the techniques presented in this thesis have been developed and a detailed description of HR, the ATF system which we build upon, is presented.

2.1 Formal methods

The failure of software systems to perform as expected can generate high losses for companies and in safety critical systems can even mean a threat to human lives. Previous experiences have shown evidence of the need for high-quality software. For instance, the flight of the Ariane 5 launcher [1] self-destroyed after just 40 seconds of its launch due to an overflow error when trying to convert 64-bits of data to 16-bits. The failure represented over 850 million dollars in losses. Another case was the Therac-25 [87], a computer-controlled radiation therapy system that overdosed six people resulting in the death of two. The new design of the Therac-25, successor of the Therac-20, contained errors which caused a failure in the interlocking system and lead to the overdose.

Formal methods are mathematical rigorous techniques used for the development and verification of software and hardware systems. They complement traditional development techniques, increasing confidence about the correctness and reliability of systems. Formal methods can be used at different stages of the development life-cycle:

- At the requirements stage, as reported in [49], for the elicitation, traceability, evolution, and other aspects of requirements engineering.
- At the design stage to write models of systems using formal specification languages such as B [2], VDM [75], Event-B [3] and Z [114].
- At the implementation stage, with techniques for code generation such as [24] and [43]

that generate JML and Ada code from Event-B, as well as languages to annotate and verify code such as Spec# [10] for C# programs, ESC/JAVA2 [26] for Java programs, SPARK [9] for (a subset of) Ada [14] programs, VCC [30] for concurrent systems written in C and Dafny [83] to verify functional correctness for the .NET platform.

- At the testing stage as described in [63], which reports the use of assertions to provide test probes. In this thesis the focus is on the development of techniques to support the design stage; in particular, refinement-based formal specification languages.

2.1.1 Formal specification

Formal specifications are mathematical descriptions of systems whose semantics are well defined and that can be subject to formal analysis, i.e. it is possible to reason about their correctness. A key aspect of formal specifications is *abstraction*, a modelling process that focuses on describing the intrinsic requirements of systems while hiding away implementation details. In other words, a formal specification describes *what* the system does rather than *how* it does it. Different types of systems can be described through formal specification; for instance, process algebras like CSP [62] and CCS [97] are used to model concurrent systems and to reason about them via the use of algebraic laws; the Z [114], VDM [75], B [2] and Event-B [3] formalisms are used to specify state-based aspects of systems; the Promela language [64] is used for modelling distributed systems, among others.

However, the development of high quality and correct models has been identified as a difficult task. In the formal methods survey presented in [119], formal specification was estimated to be the phase with the higher increase in the development time, while in [113] it was identified that choosing the right set of abstractions was the main barrier when writing formal models. As described in [77], techniques such as *decomposition* and *refinement* have been developed in order to aid formal modelling. Decomposition allows the verification of a system through the individual verification of its sub-components while refinement enables the gradual verification of systems through the use of incremental steps. Here we focus on refinement.

2.1.2 Refinement

Refinement[1] is a technique used to model systems at different levels of abstraction. Its main purpose is to handle the complexity of large systems through the gradual introduction of steps that are verified by proof. Starting from an abstract representation of a system, details are added incrementally in the search for a more concrete representation which is closer to

[1] Refinement is sometimes called *reification* [75].

implementation. As described by Abrial in [3], refinement can be achieved via two complementary techniques:

- *Vertical refinement:* known as well as *data refinement*, makes reference to the refinement of data types, i.e. the transition from abstract data types to concrete data structures. The rationale for the transition is usually specified through *gluing invariants* as in the Event-B formalism or *retrieve functions* as in VDM. The consistency of the transformation is verified by proving that the concrete operations preserve that rationale.

- *Horizontal refinement:* refers to refinement steps in which new requirements or more detailed functionality are introduced into the model. The correctness of each step is verified by proving that the behaviour at the concrete level is consistent with the behaviour at the abstract level.

Additionally, refinement can be achieved via two main approaches. Firstly, the *rule-based approach*, which uses predefined rules whose correctness has been previously verified. The most notable example is the technique proposed by Carroll Morgan [99] where a set of basic refinement transformation rules are introduced. Secondly, the *posit-and-prove approach*, which allows users to explore their own refinements but a formal proof is then required in order to determine the correctness of the steps. Formalisms such as VDM [76], B [2] and Event-B [3] implement this style of refinement. The techniques developed in this thesis are tailored for the posit-and-prove approach.

There are several techniques that have been proposed for the enhancement of the posit-and-prove approach. In general, these can be seen as incorporating rule-based features. For instance, BART [107] is a tool for classical B, which uses a series of small rules to automate the refinement of a specification into an implementation by targeting specific elements of a model at a time; for instance, transforming a variable. The application of these rules may yield more than one alternative refined model. At present these rules are not formally verified, i.e. it is the responsibility of the user to prove the correctness of each rule application. Another technique is the ZRC refinement calculus for Z [25], which introduces a refinement method for Z that follows the style of Morgan's work and that supports procedures, recursion, and data refinement. ZRC is completely formalised and therefore its transformation rules yield correct refinement steps. Pattern-based methodologies have also been explored in [66, 67, 6, 60]. In their methodologies a pattern is represented by an abstract and a concrete model that specify the structure of the refinement. The patterns are then applied by matching the abstract part of the pattern with the model of the user, and automatically producing the refinement steps alongside the proofs. The aim of all these techniques is to reuse common refinement steps and increase the productivity by automatically producing a proved refinement

step.

The techniques developed in this thesis are focused on refinement-based formalisms, and in particular on Event-B. A brief description is given next about some relevant formalisms that are based on refinement:

- *VDM:* VDM [75], which stands for Vienna Development Method, is a formalism originated in the 70s at the IBM laboratory in Vienna. VDM is used for the development and verification of software systems. A VDM specification is a mathematical model composed of a state and a set of operations which are defined through pre- and post-conditions. Refinement is implemented in VDM via data reification and operation decomposition. A *retrieve function*, which relates the abstract data types and their corresponding concrete types, is used in VDM to explain data reification.

  VDM has been extended to VDM++ [42], which supports the modelling of object-oriented and concurrent systems. A large number of tools are available for the development of systems in VDM; for instance, the Overture development environment[2] for VDM, which includes model editing, syntax and type checking, debugging and proof obligation generation among others. VDM has been widely used in the industry; one of its most recognised applications is the development of compilers, in particular the first European Ada compiler [29].

- *The Z notation:* Z [114] is a formal specification language first introduced by Abrial, Schuman and Meyer in the early 80s [7]. The Z notation is based on mathematical constructs used in set theory and first order predicate logic. The state of a system in a Z specification is represented by global variables, predicates are use to express the types of variables as well as invariants, and operations are structured through schemas. Schemas are atomic actions expressed in terms of inputs and outputs as well as pre- and post-conditions.

  Refinement is possible in Z via ZRC. Z has also been extended to allow the specification of complex systems by introducing object-oriented constructs and notions such as classes, inheritance and polymorphism [82]. Tool support is also available for the development of Z specifications; this includes test case generation tools, model checking, animation and type-checkers, among others.

- *The B method:* B [2] is a formalism for the specification, design and coding of software systems. A B specification is composed of variables that describe the state of the system, invariants which describe properties of the variables that must always hold, and a set of operations that define changes in the state. B specifications are built by

[2] See `http://www.overturetool.org/`.

means of refinement of *abstract machines*. An abstract machine specifies the basic requirements of the system and is subsequently refined all the way to implementation via *refined machines*, which refine an abstract or a refined machine; and an *implementation machine*, which represents the last model from which code can be automatically generated. The verification of B developments is achieved through the generation of proof obligations, which are used to check the correctness of the model against the invariants and the consistency between different levels of refinement.

The B method has been successfully applied to industrial projects, one of the most successful applications is its use in the development of Line 14 of the Paris metro. The B method is also supported by a set of tools such as Atelier B[3], which enables the development of industrial-sized developments, and BART [107], a tool that automates refinement in B.

- *Event-B:* Even-B [3] is a formalism used for the modelling of discrete event systems. An Event-B development is structured into models and contexts. A context describes the static part of a system, i.e. constants and their axioms, while a model describes the dynamic part; i.e. variables, invariants and events. Event-B promotes refinement-based formal modelling, where each step of a development is underpinned by formal reasoning. That is, each refinement step generates proof obligations that must be discharged in order to prove the correctness of the step. A detailed description of Event-B is provided in Section 2.2.

2.1.3 Invariant Discovery

The work on invariant discovery can be classified into two main lines of research: *static analysis* and *dynamic analysis*. Static techniques, such as data flow analysis, explore the code of the system in order to reason about it via symbolic execution. This approach is for instance applied by Givan [51], where postconditions are obtained from the analysis of an operations' preconditions and their semantics. Another technique for static analysis is proposed in [73], in which invariants are detected at the level of requirements by analysing a state machine model associated to the requirements specifications. Shape analysis also applies static analysis by inferring invariants from graph structures that model memory access, an example of this is [110]. Term synthesis has also been applied for the discovery of loop invariants by combining program analysis and proof planning [91]. Other approaches to invariant discovery have been based on proof planning; for instance [71, 115] in which the discovery is driven by the failure of rippling.

[3] See `http://www.atelierb.eu/en/`

Dynamic techniques, on the other hand, examine variable traces from test runs and report properties that are observed to be true over such executions. Current dynamic approaches, such as the Daikon system [44], infer invariants by examining existing templates against execution traces, where a potential invariant is one that is true for all states in the trace. Special purpose techniques have also been developed; for instance, value profiling [23], which addresses the detection of constant and near-constant properties, and [116] which allows the discovery of ordering relationships within the Spin model checker. Machine learning techniques also provide approaches to concept extraction from data sets. An example is Inductive Logic Programming (ILP) [100], which has been used to discover loop invariants [18]. A key aspect of machine learning techniques like ILP is that they require positive and negative examples in the data set in order to generate relevant invariants.

The invariants discovered by static approaches are true for all program runs, while for dynamic approaches not all the reported invariants turn out to be true. However, dynamic approaches are more flexible since they focus on behaviour instead of code structure.

2.1.4 Automated Reasoning

Formally verifying the correctness of systems is a complex task which requires the support of automatic tools. Automated reasoning is the study of techniques to automate formal reasoning. Particular development effort has been carried out in two subareas of automated reasoning: namely, *theorem proving* and *model checking*. A brief description of these techniques is provided in the following sections.

2.1.4.1 Theorem Proving

Theorem proving techniques are used to verify if a mathematical statement follows logically from a set of axioms through the application of inference rules; i.e. if it is a theorem of the theory under consideration. The conjecture, axioms and inference rules are all written as predicates of a logic that enables the verification. Theorem proving then involves the use of search control techniques to automate the search for mathematical proofs.

The first attempt for a theorem prover that showed that the automatic search for proofs in a computer was possible was by Paul Gilmore in 1960 [50]; however, the system could only perform proofs of trivial theorems. A subsequent effort was the Logic Theory Machine [101], a heuristic-based system that built proofs for a small subset of propositional logic. Further advances were made in the following years; the main attempts can be categorised into two complementary type of systems: *machine-oriented* automated theorem provers and *human-oriented* theorem provers. Machine-oriented provers are those based upon computer-oriented inference systems, which are able to solve mathematical problems through the application

of automatic proof steps. These provers either provide a complete proof or no proof at all. The most notable invention in this field is resolution [109], a technique that performs proofs by contradiction. Furthermore, systems that incorporate more powerful constructs, called *tactics*, were developed. These are small programs that manipulate a proof by applying sequences of steps simultaneously. The first work on tactic-based theorem proving was in the LCF project [54]. However, as shown by Gödel's incompleteness theorem, proving all theorems of a sufficiently powerful theory is undecidable. Human-oriented theorem proving systems attempt to address this limitation. While machine-oriented systems rely on generic proof strategies, human-oriented systems focus on domain specific heuristics. Although their application does not guarantee success, if the fail, such approaches provide insight to a user in the search for a proof. The most prominent work on this area is proof planning [19], which divides reasoning into two levels: a planning level, which heuristically reasons about a proof, and an execution level, which applies a tactic that executes the plan. Example of planning-based techniques are rippling [20] and proof critics [68], which use meta-level reasoning and failure analysis to guide the search for proofs.

A variety of theorem provers are currently available. For instance, PVS [103] and ACL2 [79], integrate verification systems that include higher-order logic and first-order logic theorem provers, respectively; Isabelle/HOL [102], an interactive theorem prover, and successor of HOL [53], also integrates a framework for proof planning: IsaPlanner [40]; Coq [16] is an interactive theorem prover for the calculus of inductive constructions; among others. Theorem proving has been used in some fields of the industry, for instance in the verification of integrated circuits of microprocessors, e.g. AMD and Intel, for flight control systems and in general for safety-critical systems.

2.1.4.2 Model checking

Model checking is an algorithmic technique to exhaustively and automatically verify if a property is satisfied by a model. The study of model checking problems started with the work of Clarke and Emerson [27] in the early 80's in concurrency and verification as an alternative to theorem proving and a fully automatic way of proving the correctness of programs. More specifically, model checking procedure takes a model $\mathcal{M}$ and a logical formula φ as input and asks the question *does* $\mathcal{M}$ *have the property specified by* φ*?*; i.e. $\mathcal{M} \vDash \varphi$?.

Unlike theorem proving, model checking procedures are always intended to be fully automatic, which is the reason why they were tailored to be used over finite state systems; however, new developments allow the verification of some infinite state systems too. The main limitation of model checking is well known: the state explosion problem. In order to tackle this problem a number of techniques have been studied, for instance, symbolic model-checking [95], partial-order model checking [52], or unfoldings [45], amongst oth-

ers. Examples of well known model checkers include: SPIN [64], a model checker for the Promela language; ProB [86], for checking B and Event-B specifications; UPPAAL [13], which is used to verify hybrid and timed systems; PRISM [80], a model checker for probabilistic systems; and SMV [28, Chapter 8], a tool for symbolic model checking.

2.2 Event-B

Event-B is a formalism used for the modelling of discrete event systems [3]. It promotes an incremental style of formal modelling where each step of a development is underpinned by formal reasoning. Event-B is an evolution of the B-method [2], and it builds upon the Action System formalism [8]. Uses set theory and first order logic as the modelling notation, refinement to handle complex systems, and proof to verify the correctness of the models and consistency between refinement steps.

As shown in Figure 2.1, an Event-B development is structured around *machines* and *contexts*. Machines represent the dynamic part of the system while contexts model the static part. A machine is said to *see* a context, i.e. machines can reference the elements of the contexts they see. Contexts are structured in terms of *carrier sets* that define the basic types in the system, *constants*, which represent the static values and *axioms* that describe properties of the constants. Machines are defined in terms of *variables*, which represent the state of the system, *events*, which are conditioned actions that define the way the state changes, and *invariants* that specify properties over the state that must always be preserved by the events.

An event takes the following general form:

$$\langle name \rangle \mathrel{\widehat{=}} \textbf{any } \langle parameters \rangle \textbf{ where } \langle guards \rangle \textbf{ then } \langle actions \rangle$$

where the *parameters* are arbitrary values received by the event, the *guards* are conditions that have to be true for the events to be triggered, and the *actions* define the changes over the variables. All machines have an *initialisation* event, which defines the initial state of a model. The initialisation event does not have parameters or guards.

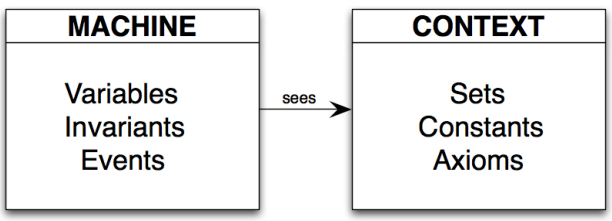


Figure 2.1: Event-B model structure.

To illustrate the basic features of Event-B, we draw upon an example of a vending machine model that dispenses various products and for which a stock count is kept. The model does not deal with money, instead it focuses on managing the stock availability for each product within the vending machine. The model is shown in Figure 2.2.

CONTEXT:
Sets PRODUCT

MACHINE:
Variables
available soldout stock products
Invariants
products $\subseteq$ PRODUCT
stock $\in$ products $\rightarrow \mathbb{N}$
partition(products, available, soldout)

Events
Initialisation
then
available := $\varnothing$
soldout := $\varnothing$
stock := $\varnothing$
products := $\varnothing$
end

Event addProduct $\widehat{=}$
any p amount
where
p $\in$ PRODUCT\products
amount $\in \mathbb{N}_1$
then
products := products$\cup${p}
available := available$\cup${p}
stock := stock$\cup${p$\mapsto$amount}
end

Event buy_1 $\widehat{=}$
any p
where
p $\in$ available
stock(p) > 1
then
stock(p) := stock(p)$-$1
end

Event buy_2 $\widehat{=}$
any p
where
p $\in$ available
stock(p) = 1
then
available := available\{p}
soldout := soldout$\cup${p}
stock(p) := 0
end

Event reStock $\widehat{=}$
any p amount
where
p $\in$ soldout
amount $\in \mathbb{N}_1$
then
stock(p) := amount
soldout := soldout\{p}
available := available$\cup${p}
end

Figure 2.2: Event-B model of a vending machine system.

The context contains a carrier set named *PRODUCT* while the machine is composed of four variables: *products, stock, available* and *soldout*. The system classifies the *products* according to their *stock* availability into sets *available* and *soldout*. The types of the variables are defined by the invariants. As can be observed, variable *products* is a subset of the carrier set *PRODUCT*, *stock* is a function that maps each product to its current stock; and *available* and *soldout* partition the products set. Note that *partition* is a primitive of Event-B, and is used here to ensure that the available and soldout sets are disjoint.

The model contains five events: *addProduct,* buy_1*,* buy_2*, reStock* and an *initialisation* event. A new type of product can be added to the vending machine through the *addProduct* event, which will also set its initial stock as well as make it available. The buy_1 and buy_2 events decrease the stock of a given product. Event buy_1 triggers when there are multiple units of a product available in the vending machine, while the buy_2 event triggers when there

is only one unit left. Additionally, the buy_2 event moves the product from the *available* set to the *soldout* set. Finally the *reStock* event sets a new stock to a product that has been soldout and makes it available.

Event-B is a language designed for the modelling of reactive and distributed systems. These type of systems are characterised for being composed of many different parts that interact with each other, that often exhibit concurrent behaviour and whose environment tend to evolve. Event-B manages the complexity of these type of systems via *Refinement*, *Decomposition* and *Generic Instantiation* [5]. Refinement promotes the evolution of systems through incremental changes, decomposition allows a development to be divided into smaller workable models and generic instantiation allows one to reuse a generic development within an outgoing one. Key to our work is the use of refinement. Next, we describe in more detail how refinement can be accomplished within the context of Event-B.

2.2.1 Refinement in Event-B

Refinement is the process of transforming an abstract model into a concrete implementation. Event-B implements stepwise refinement, that is progressively making an abstract specification more precise through a series of incremental steps. Each step creates a more concrete model which is a refinement of the previous one and must be verified through the use of proof. This modelling style facilitates the verification of large complex systems by focusing on each refinement step at a time.

In an Event-B model, contexts are *extended* while machines are *refined*. This is illustrated in Figure 2.3. A context is extended by preserving the elements from the abstract context, i.e. carrier sets and constants, and adding new ones at the concrete level. The new constants are defined through the use of axioms that represent their properties.

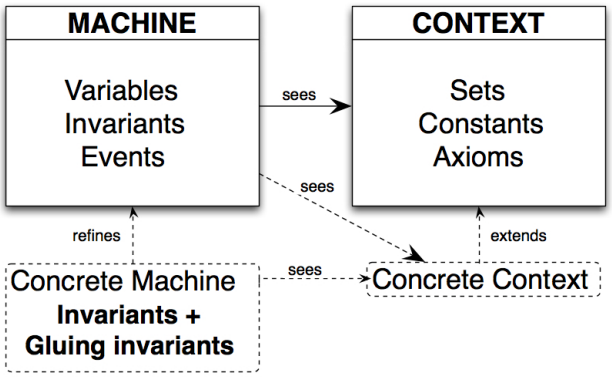


Figure 2.3: Event-B refinement structure.

In the refinement of a machine, the state of the concrete machine can be extended or

modified by replacing abstract variables with new variables in the refinement. As with the abstract machine, the concrete machine includes invariants that handle its variables. However, the refined machine includes a special invariant that relates the states of the abstract and concrete machines, that is the *gluing invariant*. This invariant is required in order to prove that the behaviour of the concrete machine is consistent with the behaviour of its abstraction. Machines can also be refined by adding new events or modifying existing ones. The refinement of existing events is verified by proving that the concrete event implies the abstract event and the gluing invariant; while the addition of new events is verified by proving that the new event hold the gluing invariant. This is extended later in this chapter.

In order to illustrate refinement in Event-B, we use the model of the vending machine introduced in Figure 2.2. In the model, the products are classified by the disjoint sets *available* and *soldout*. At the concrete level, the representation is changed to a total function *status* that maps products to their availability. The state of products is represented by an enumerated set *STATE* whose members are the constants *AVAILABLE* and *SOLDOUT*. The refinement step is shown in Figure 2.4.

As can be observed, the refinement affects the context as well as the machine. Note also that where an abstract event is refined, the keyword **refines** is used to indicate the *refining* event at the concrete level. Observe that where applicable, the events have been modified, replacing the abstract disjoint set by a function check.

It is important to note that the refinement presented in Figure 2.4 is only provable if the required gluing invariants are in place. In this case, the gluing invariants must relate the disjoint sets from the abstract model with the new function at the concrete level. The gluing invariants from the model are:

$$\text{available} = \text{status}^{-1}[\{\text{AVAILABLE}\}] \tag{2.1}$$

$$\text{soldout} = \text{status}^{-1}[\{\text{SOLDOUT}\}] \tag{2.2}$$

These invariants state that the abstract sets *available* and *soldout* can be obtained from the inverse of the function *status* evaluated over the enumerated sets *AVAILABLE* and *SOLDOUT*.

2.2.2 Verifying Event-B developments

Event-B developments are verified through the use of Proof Obligations (POs). A PO is a sequent of the form:

$$H \vdash G$$

CONTEXT:
Sets
STATE
Constants
AVAILABLE SOLDOUT
Axioms
partition(STATE,{AVAILABLE},{SOLDOUT})

MACHINE
Variables
products status stock
Invariants
status ∈ products→STATE
available = status^{-1}[{AVAILABLE}]
soldout = status^{-1}[{SOLDOUT}]

Events

Initialisation
then
products := ∅
status := ∅
stock := ∅
end

Event addProduct ≙
refines addProduct
any p amount
where
p ∈ PRODUCT\products
amount ∈ $\mathbb{N}_1$
then
products := products∪{p}
status := status∪{p↦AVAILABLE}
stock := stock∪{p↦amount}
end

Event buy$_1$ ≙
refines buy$_1$
any p
where
p ∈ products
status(p) = AVAILABLE
stock(p) > 1
then
stock(p) := stock(p)−1
end

Event buy$_2$ ≙
refines buy$_2$
any p
where
p ∈ products
status(p) = AVAILABLE
stock(p) = 1
then
status(p) := SOLDOUT
stock(p) := 0
end

Event reStock ≙
refines reStock
any p amount
where
p ∈ products
status(p) = SOLDOUT
amount ∈ $\mathbb{N}_1$
then
status(p) := AVAILABLE
stock(p) := amount
end

Figure 2.4: Refinement step of the vending machine model.

where H represents the set of *hypotheses* and G represents the *goal* to be proved.

There are two different types of POs associated with Event-B developments: POs relating to a single level in the refinement chain and POs relating to an abstract model and its refinement. Next we give a brief description of each of these POs. We use E to denote an Event, $\overline{M}$ to denote a list of abstract and concrete models related by a refinement step, $BAP(E)$ to denote the before-after predicate associated to an event E and x' to denote the after value of

element x (where x denotes a variable, an invariant or a variant).

Invariant preservation PO (INV): this PO verifies that an invariant is preserved by an event before and after its execution. For an invariant inv, the INV PO has the following form:

$$Axioms(\overline{M}) \land Invariants(\overline{M}) \land Guards(E) \land BAP(E) \vdash inv'$$

Feasibility PO (FIS): this PO is used to ensure that a non-deterministic action is feasible; i.e. that there exists an after-value for which the before-after predicate of the non-deterministic action is true. The FIS PO has the following form:

$$Axioms(\overline{M}) \land Invariants(\overline{M}) \land Guards(E) \vdash \exists\, v'.BAP(E)$$

Guard strengthening PO (GRD): this PO ensures that the guard of a refined event implies the guard of the abstract event it refines. For an event E_1 that is refined by event E_2, the GRD PO has the following form:

$$Axioms(\overline{M}) \land Invariants(\overline{M}) \land Guards(E_2) \land Witnesses(E_2) \vdash guard(E_1)$$

Note that a GRD PO is generated for each guard of the abstract event.

Simulation PO (SIM): this PO verifies that the actions of a refined event simulate the same behaviour than the abstract event it refines. For an event E_1 that is refined by event E_2, the SIM PO has the following form:

$$\textit{Axioms}(\overline{M}) \land \textit{Invariants}(\overline{M}) \land \textit{Guards}(E_2) \land \textit{Witnesses}(E_2) \land \textit{BAP}(E_2) \vdash \textit{BAP}(E_1)$$

Variant PO: this PO ensures that under the guards of each convergent or anticipated event, a variant holds. This PO has different variations according to the type of the variant, that is: natural numbers or sets.

- Variant PO to ensure a variant is a natural number (NAT).

$$Axioms(M) \land Invariants(M) \land Guards(E) \vdash variant \in \mathbb{N}$$

- Variant PO to ensure a variant is a finite set (FIN).

$$Axioms(M) \land Invariants(M) \land Guards(E) \vdash \textit{finite}(variant)$$

- Variant PO to ensure that the numeric or set variant is decreased by each convergent or anticipated event (VAR).

  For convergent events:

  $\mathit{Axioms}(M) \land \mathit{Invariants}(M) \land \mathit{Guards}(E) \land \mathit{BAP}(E) \vdash \mathit{variant} < \mathit{variant}'$

  $\mathit{Axioms}(M) \land \mathit{Invariants}(M) \land \mathit{Guards}(E) \land \mathit{BAP}(E) \vdash \mathit{variant} \subset \mathit{variant}'$

  For anticipated events:

  $\mathit{Axioms}(M) \land \mathit{Invariants}(M) \land \mathit{Guards}(E) \land \mathit{BAP}(E) \vdash \mathit{variant} \leq \mathit{variant}'$

  $\mathit{Axioms}(M) \land \mathit{Invariants}(M) \land \mathit{Guards}(E) \land \mathit{BAP}(E) \vdash \mathit{variant} \subseteq \mathit{variant}'$

Witness feasibility PO (WFIS): this PO verifies that for an event for which a witness has been proposed, there is indeed a value that holds the witness. For an event E, the WFIS PO has the following form:

$$\mathit{Axioms}(M) \land \mathit{Invariants}(M) \land \mathit{Guards}(E) \land \mathit{BAP}(E) \vdash \exists\, w.\mathit{Witness}(E) = w$$

Theorem PO (THM): this PO verifies that a theorem follows from the invariants and/or axioms of the model. Depending if the theorem is in the context or in the machine the THM PO has the following forms:

Context theorem: $\mathit{Axioms}(M) \vdash \mathit{theorem}$

Machine theorem: $\mathit{Axioms}(M) \land \mathit{Invariants}(M) \vdash \mathit{theorem}$

Guard strengthening for merged events PO (MRG): this PO ensures that the guard of a refined event that merges two or more events implies the guards of the events it merges. For a set of events $E_{A_1}...E_{A_n}$ which are merged at the concrete level by event E_2, the MRG PO has the following form:

$$\mathit{Axioms}(M) \land \mathit{Invariants}(M) \land \mathit{Guards}(E_2) \vdash \mathit{Guards}(E_{A_1}) \lor ... \lor \mathit{Guards}(E_{A_n})$$

Well-definedness PO (WD): this PO ensures that all predicates and expressions within an Event-B development are well defined. The WD PO has several forms depending on the potentially ill-defined expression. For instance, for a partial function, the expression $F(x)$ has the well-definedness condition $x \in dom(F)$.

In order to illustrate the generation of POs within Event-B, two POs associated with event

buy$_2$ of the model of the vending machine (Figure 2.4) are presented. A guard strengthening PO is shown in Figure 2.5(a). This PO specifies that the refined event must imply the guard $p \in available$ – that is, prove that the abstract guard follows from the concrete guard $status(p) = AVAILABLE$. On the other hand, an invariant PO is shown in Figure 2.5(b), which verifies that the gluing invariant $available = status^{-1}[\{AVAILABLE\}]$ is preserved after the execution of the event – in this case it must be proved that changing the status of p form *AVAILABLE* to *SOLDOUT* at the concrete level is equivalent to removing p from the set $available$ at the abstract level. Both POs are trivial and are discharged automatically.

p ∈ sellProducts	available = status^{-1}[{AVAILABLE}]
status(p) = AVAILABLE	p ∈ products
stock(p) = 1	status(p) = AVAILABLE
⊢	stock(p) = 1
p ∈ available	⊢
	available\{p} = (status⩤{p↦SOLDOUT})$^{-1}$[{AVAILABLE}]
(a) GRD PO.	(b) INV PO.

Figure 2.5: Proof obligations associated to event buy$_2$ in the vending machine model.

2.2.3 Rodin

Rodin [4] is a platform implemented on top of the Eclipse environment[4] for the development and verification of Event-B specifications. Rodin allows a developer to reason about a model by giving instant feedback about its correctness. This is achieved by automatically generating and discharging POs, which allows the integration of reasoning as part of the modelling task during the development of Event-B models. Furthermore, discharging POs may not always be automatic; depending on the model, user interaction may be needed to discharge a PO. The close interplay between modelling and reasoning provided by the Rodin toolset facilitates the identification of problems when a PO fails to verify. It is important to stress that Rodin is not used to run programs but to reason about models at the design stage.

The Rodin tool chain is composed of three main components:

The static checker (SC): analyses a model developed in Event-B in order to find syntax and type errors.

The proof obligation generator (POG): automatically generates the POs that must be verified for a given Event-B model – the different type of POs associated with Event-B were described in Section 2.2.2. The POG does not perform proofs, it only carries out simple rewritings within a PO sequent.

[4]http://www.eclipse.org/

The proof obligation manager (POM): handles the POs' status as well as the associated proof tree for each PO. It works automatically alongside the automatic Rodin provers or interactively with the user and external provers – since all POs are represented as sequents in predicate calculus, different external provers for predicate calculus can be used through Rodin.

Rodin provides similar functionalities to those provided by tools used for programming, in which tasks are performed automatically in the background. This facilitates and improves the modelling experience for Rodin users. Among the characteristics offered by Rodin are:

- instant feedback when a change has been made to the model; i.e. syntax errors, inconsistent types, etc.,
- automatic generation and verification of POs when a model is saved to the disk (no need of compilation processes),
- error traces,
- management of a schema of colours for reserved words (which make the models more readable),
- templates for the creation of Event-B basic elements; i.e. events, variables, etc.

Furthermore, Rodin is composed of two main graphical interfaces: the *modelling perspective* and the *proving perspective*. The former allows users to edit their models, see problems related to them and explore the structure and status (POs) of their Event-B projects. The latter allows the user to explore the proof tree associated to a PO[5] as well as to perform interactive proofs. Moreover, because Rodin is built on the Eclipse platform, new functionalities can be provided through the addition of plug-ins. This flexible architecture contributes to the improvement and extensibility of the tool as well as to the formation of a bigger community working around Event-B. We mention some of the plug-ins available in Rodin which illustrate different aspects of the toolset that have been extended :

- UML-B [112]: graphical front-end for the modelling of Event-B systems as UML-like diagrams. Currently contains modelling and refinement of systems with class and state machine diagrams.
- ProB [86]: provides animation and model checking capabilities for Event-B models.
- ProR [72]: provides requirement traceability between an Event-B model and the natural language requirements associated to the model.
- Atelier B provers [6]: provide additional capabilities of automated theorem proving (not contained in the Rodin provers).
- B2Latex[7]: exports Event-B models as Latex documents.

[5] A PO may not have a proof tree associated with it if an external prover has been used in order to discharge the PO (or part of it).

[6] `http://www.atelierb.eu/en/`

[7] `http://wiki.event-b.org/index.php/B2Latex`

Currently the development of new plug-ins for the Rodin platform is growing[8]. We describe in more detail the ProB simulator and model checker, which is key for our research.

2.2.4 ProB

ProB is a simulator and model checker for the B method implemented in Prolog. It supports automated consistency checking of B machines through model checking and constraint-based checking as well as the simulation of B machines. The consistency checking and simulation capabilities of ProB provide a way of facilitating the verification of B models. That is, ProB can provide counter-examples when a model is not free of errors. This means that current unproven POs are indeed not provable and therefore, the user does not have to spend time trying to discharge proofs that are false.

The model checker verifies if an invariant of the model is violated by finding a sequence of operations that starting from a valid initialisation reaches a state where the invariant does not hold. If this situation is found, the model checker provides the shortest trace that leads to the error within the states that have been explored. In order to do exhaustive model checking over B models, the model should have a finite set of states, and the size of sets and the bounds of integer variables are limited. ProB can also be used over models with infinite set of states. In this case the model checker will finish when an error is found or when the system runs out of memory.

The constraint-based checker verifies if an individual operation can violate an invariant independently of the initial state. This is achieved by setting up constraints that ensure that the starting state satisfies the invariant. Then, the operation is applied and new constraints are set up and evaluated to determine if the invariant has been violated. If this is the case, concrete values for the variables in the model are instantiated and a counter example is provided.

The underlying mechanism of the constraint-based checker is more complex and requires more resources than the model checker. However, for situations when the checking can be focused on a single operation; for instance, when adding or modifying an operation of a machine previously proved, the constraint-based checker may be preferred. Also, as constraint-based checking can be enabled after finding one state that satisfies the invariant, it can be executed when an exhaustive search is too costly.

ProB can also be used as a simulation tool to explore and debug a model. This functionality enables the identification of errors, showing inconsistencies between models, invariant violations, references to undefined expressions. When an error is identified during a simulation, ProB provides the concrete values of the variables for which the model fails.

[8]The complete list of past and current plug-in developments can be found at `http://wiki.event-b.org/index.php/Rodin_Plug-ins`.

ProB is available for Rodin in the form of a plug-in that can be installed through the software update option provided by Eclipse. The plug-in provides model checking and simulation capabilities for Event-B models. We use simulation for our research. The simulation of Event-B models can be performed step by step or through a sequence of random steps (the number of steps to be performed must be specified by the user). The simulator provides the user with information about the current state of the system, the history of steps (events) applied, the events currently enabled and any errors found within the model. For each enabled operation ProB provides a set of possible values that can be used to instantiate the parameters of the operation (if any are needed). Limits for the size of sets and integer range must be provided by the user; otherwise the default values are used.

The ProB plug-in for Rodin also allows the simulation of various levels of refinement simultaneously. This process requires one to load into ProB a machine and all its ancestor machines–or a number of ancestor machines specified by the user–and all the contexts seen by them. Then, ProB produces an internal representation of each event of the most concrete machine. This representation contains the list of all abstract events related to the concrete event. During simulation, the state is composed of all the variables of the machines loaded into ProB and the invariant consists of the conjunction of the invariants of each machine. The steps of the simulation, as described in [58], are:

1. Find valid values for all constants.
2. At each step ProB executes an event of the most concrete machine as follows:
   (a) Find valid values for the parameters of the event.
   (b) Execute each action of the event. An error is found at this stage if: i) the predicate associated with a non-deterministic action does not hold (FIS PO fails), or ii) the value assigned to a variable cannot be assigned in the abstract event (SIM PO fails).
   (c) If the event has witnesses, there must exist valid values for the witnesses; otherwise an error is reported (WFIS PO fails).
   (d) If more than one abstract event is associated with the concrete event, one of them is chosen and evaluated as in step 2a. If valid values are found, the simulator continues recursively with step 2b; otherwise, it tries another of the abstract events. If none of the abstract events hold, an error is produced since the guard of the most concrete event is weaker than its abstractions (GRD PO fails).

As can be observed, ProB can detect different errors associated with refinement. For instance, guard weakening, witnesses that are not feasible, inconsistent behaviour. The same algorithm is applied by the model checker.

Simulation of multiple levels in the refinement is a key feature of our work on invariant

discovery. This functionality allows us to simulate a machine and its immediate abstraction in order to observe patterns in their behaviour and discover missing invariants.

2.3 Automated Theory Formation and HR

Artificial Intelligence (AI) has been used for the development of tools and techniques that automate aspects of reasoning in mathematics and systems development. For instance, computer algebra systems, theorem provers, constraint solvers and SAT solvers are used to assists mathematicians as well as for hardware and software verification. ATF is a novel machine learning approach to automated reasoning which explores mathematical domains in order to construct theories about them. The theory formation process consists of exploring background information that describes a domain in order to invent new concepts from the old background information and hypothesise relationships within the newly generated concepts.

The term *Automated Theory Formation* was first introduced by Lenat in [85] where he presented the Automated Mathematician (AM) system, the first system that performed both concept invention and conjecture inference to relate the invented concepts[9]. Subsequent work on ATF systems includes Eurisko [84], the successor of AM, which attempted to generate new heuristics with better applicability throughout the theory formation process; Cirano [56], which focused on constraining some implementation aspects of Eurisko in order to make the theory formation cycle more efficient; the ARE system [111], which extended concept invention through the introduction of functional transformation; HR [32], which uses an iterative application of general purpose production rules for concept invention. Consecutively, ATF was further developed by improving and developing new techniques as to how concepts were invented and conjectures were identified, and by applying it within new mathematical domains such as graph theory and plane geometry, among others. More recently, Lakatos's [81] characterisation of ways in which mathematicians respond to counterexamples and use them to evolve concepts, conjectures and proofs form the basis of the theory formation system HRL [104] and TM [38]; IsaCoSy [74] and IsaScheme [98] support the discovery of theorems within the context of mathematical induction, the CORE system [91] supports term synthesis within the context of separation logic, among others.

As it has been described, ATF was developed with the aim of exploring mathematical domains; however, new applications have emerged through the time. ATF has being studied within Computational Creativity [33] as a case study of the application of a model that describes and categorises the creativity of programs. Moreover, ATF has been used for the invention of games [12] and the automatic generation of puzzles [31]. In the former, data

[9]However, the work carried out by Lenat in the AM system has been criticised by AI researches. For instance, through a case study of Lenat's work [108], Ritchie and Hanna discovered the use of heuristics implemented for specific purposes contrary to claims of the use of a generic methodology.

obtained from playing simple physical games is explored in order to invent mixed reality games, while the latter extends the HR system by adding a categorisation of three types of puzzles, enabling HR to generate puzzles about theories being analysed.

The general process of theory formation involves three iterative aspects:

1. *The invention of concepts that are generated from old ones.* New concepts are invented by performing calculations or applying transformations over the existing concepts. For instance, applying arithmetic operations to numeric concepts, adding elements to build more complex concepts, such as planes in geometry, combining concepts with the same types.
2. *Identification of empirical relationships between concepts.* In this step, the theory formation engine searches for patterns that appear within the newly generated concepts and the existing concepts. Different techniques are used to hypothesise these relationships; for instance, observing that a value is constant within a concept, observing that two concepts have the same values.
3. *Proving and disproving the identified relationships*. The validity of the discovered potential relationships are usually verified through the use of theorem provers and model generators that search for counterexamples.

For the work presented in this thesis we used the HR system developed by Colton [32]. There are two main reasons we selected the HR system for invariant discovery of Event-B models. Firstly, the input format of the background information is based upon examples. This is convenient for the work on invariant discovery since examples can be easily obtained in Event-B through the simulation of the model. Moreover, different from other theory formation systems such as ILP [100], which are also based on examples, HR does not require positive and negative examples in order to form theories. Secondly, the transformation technique applied by HR to invent new concepts allows Event-B-like relationships to be formed. Below we consider the HR system in more detail.

2.3.1 The HR system

Colton's machine learning system HR[10] [32] performs descriptive induction, which aims at finding interesting relationships in unclassified relational data, to form a theory about a set of objects of interest which are described by a set of core concepts. This is in contrast to predictive learning systems which are used to solve the particular problem of finding a definition for a target concept. HR constructs a theory by finding examples of objects of interest, inventing new concepts through the use of production rules (PRs), making plausible

[10]HR is named after mathematicians Godfrey Harold Hardy (1877 - 1947) and Srinivasa Aiyangar Ramanujan (1887 - 1920). A website for HR can be found in `http://www.doc.ic.ac.uk/~sgc/hr/`. To download a more recent version of HR an its documentation please contact its creator: Simon Colton.

statements relating those concepts, evaluating both concepts and statements, and proving or disproving the statements.

HR has been used for a variety of discovery projects, including mathematics and scientific domains. It has been particularly successful in number theory [35], algebraic domains [96] and constraint solvers [36, 105].

HR requires two inputs: domain information, which contains the building blocks of the theory (the background concepts and objects of interest); and macro information, which contains instructions for the way in which the theory should be constructed (e.g. which production rules are to be used, which arguments they will take, a weighted sum for the interestingness measures).

Next, we present a brief description of HR's theory formation process using a running example taken from [32] to illustrate it.

2.3.1.1 Representation of concepts

The objects of interest are the entities which a theory discusses. For instance, in set theory the objects of interest are sets, in number theory they are integers. Unary concepts are used to represent the objects of interest in HR. In this thesis, we categorise these as concepts of type 1 (T1). These concepts enumerate each object of interest within a theory. For instance, the unary concept of the integer numbers from 1 to 10 is shown in Figure 2.6.

integer(A)
1
2
3
4
5
6
7
8
9
10

Definition:
integer(A)

Figure 2.6: Unary concept representing the integers from 1 to 10.

Note that a concept is represented as a data table (or table of examples). This representation is crucial for the application of the PRs as it will be shown later. Furthermore, concepts have a definition; i.e. a signature, that specifies the type of the arguments of the concept and the relations between these arguments. For the concept presented in Figure 2.6 the definition is very simple: *integer(A)*, which states that *A is of type integer*. These definitions become more complex through the application of the PRs. Within HR, a concept also has an associated categorisation and measures of interestingness – categorisations group the examples

of a data table which have the same value, while the measures of interestingness evaluate general properties of concepts; e.g. applicability and novelty. Other type of concepts are those that define features about the objects of interest. These concepts can be provided by the user as part of the background information or developed through the theory formation process by the application of the PRs. We categorise the former as concepts of type 2 (T2) and the latter as concepts of type 3 (T3). For instance, the concept that represents the divisors of the integers from 1 to 10 is shown in Figure 2.7. This is a T2 concept provided by the user where B represents a divisor of A.

divisors(A,B)	
integer	divisor
1	1
2	1
2	2
3	1
3	3
.	.
.	.
10	1
10	2
10	5
10	10

Definition:
divisors(A, B) : *integer*$(A) \wedge$ *integer*$(B) \wedge$ *divisors*(A, B)

Figure 2.7: Concept representing the divisors of the numbers from 1 to 10.

Note that for this type of concept, the data table is a function that maps an object of interest to a truth value or a set of objects. For instance, for the data table shown in Figure 2.7, applying the function to the integer 10 yields: $f(10) = \{1,2,5,10\}$. Moreover, note that all concepts in HR are expressed either as unary predicates or as binary relations.

For the purposes of the invariant discovery work carried out in this thesis, we further distinguish between concepts that are provided by the user, what we call *core concepts*, or developed by HR through the use of the PRs, what we call *non-core concepts*. Consequently, *T1* and *T2* concepts represent *core concepts*, while *T3* concepts represent *non-core concepts*.

2.3.1.2 Concept generation

Colton [32] observed that it is possible to gain an understanding of a complex concept by decomposing it via small steps into simpler concepts. Based on this, he defined production rules which take in concepts and make small changes to produce further concepts. Each production rule is generic and works by performing operations on the content of one or two input data tables and a set of parameterisations in order to produce a new output data

table, thus forming a new concept. The production rules and parameterisations are usually applied automatically according to a search strategy which has been entered by the user, and are applied repeatedly until HR has either exhausted the search space or has reached a user-defined number of theory formation steps to perform. Currently, HR has three types of PRs: *nullary rules*, *unary rules* and *binary rules*. A brief overview of the available PRs is given below along with examples of their application. HR performs internal processes for renaming concept arguments and in some cases intermediate concepts are also required with the application of a PR. We do not get into that level of detail here but refer the reader to [32, 37] where these processes are described.

Production rules:

1. Nullary rules: these PRs receive as input a T1 concept and generate a new unary concept which represents a subset of the examples of the original concept.

the entity-disjunct rule: singles out examples, given as parameters of the rule, from the original data table in order to invent concepts that represent constants of the background domain. The output data table contains the constants extracted from the original concept. For instance, applying this PR to concept *integer(A)* (see Figure 2.6) with parameter $\langle 3 \rangle$ produces the following concept definition:

$$\textit{newC}(A) : \textit{integer}(A) \wedge A = 3$$

2. Unary rules: these PRs receive one concept as input and generate a new concept from it. The concept of divisors presented in Figure 2.7 will be used to illustrate the application of the unary PRs. Remember the definition of this concept is:

$$\textit{divisors}(A, B) : \textit{integer}(A) \wedge \textit{integer}(B) \wedge \textit{divisors}(A, B)$$

where B is a divisor of A.

the exists rule: removes specified arguments from a concept definition. The output data table contains the examples of the arguments that are kept in the concept definition. For instance, applying this PR with parameters $\langle 1 \rangle$ produces the following concept definition:

$$\textit{newC}(A) : \textit{integer}(A) \wedge \exists\, B(integer(B) \wedge \textit{divisors}(A, B))$$

This parameterisation removes argument in position 1 from the original concept

definition; resulting in a new concept which contains the integers that have associated divisors.

the match rule: finds tuples within a concept in which the examples of a specified set of arguments are equal. The output data table merges the duplicated arguments into one in the new concept definition. For instance, the application of this PR with parameters $\langle 0, 0\rangle$ returns the following concept definition:

$$\mathit{newC}(A) : \mathit{integer}(A) \wedge \mathit{divisors}(A, A)$$

which stands for the concept of integers that are divisors of themselves. The parameterisation $\langle 0, 0\rangle$ specifies that the argument in position 0 is left the same and the argument in position 1 is matched with the one in position 0. Note that the redundant arguments are removed from the resulting concept; i.e. the output datatable contains only one instance of the matching parameters.

the equals rule: this PR performs the same transformation as the *match* PR; however, here all the arguments are left in the new concept definition and the equality is expressed by adding the predicate *equals*(a_1,...,a_n) to the definition of the new concept – where a_i are the arguments that are equal. For instance, applying this PR with parameters $\langle 0, 1\rangle$ produces the following concept definition:

$$\mathit{newC}(A, B) : \mathit{integer}(A) \wedge \mathit{integer}(B) \wedge \mathit{divisors}(A, B) \wedge A = B$$

which also stands for the concept of integers that are divisors of themselves; however, the resulting data table keeps all the arguments from the original concept.

the split rule: extracts the list of tuples from the data table of a concept for which a set of specified arguments are equal to some given values. The output data table contains the tuples from the original data table that meet the conditions specified in the parameterisation. For instance, applying this PR with parameters $\langle 1 = 2\rangle$ produces the following concept definition:

$$\mathit{newC}(A) : \mathit{integer}(A) \wedge \mathit{integer}(2) \wedge \mathit{divisors}(A, 2)$$

The new concept results in the list of integers for which 2 is one of their divisors. The parameterisation $\langle 1 = 2\rangle$ specifies the output data table should contain only tuples from the original concept for which the value of the argument in position 1 is equals to 2.

the size rule: counts the number of times that the examples of a given set of arguments are repeated within a concept data table. The output data table contains

the distinct occurrences of the given set of arguments and the number of times they occur within the original data table. For instance, the application of this PR with parameters $\langle 1 \rangle$ produces the following concept definition:

$$\textit{newC}(A, B) : \textit{integer}(A) \wedge \textit{integer}(B) \wedge B = |\{C : \textit{integer}(C) \wedge \textit{divisors}(A, C)\}|$$

This concept counts the number of divisors for each integer in the background domain. The parameterisation $\langle 1 \rangle$ specifies that the argument to be counted is the one in position 1 from the original concept data table.

the linear-constraint rule: given a concept for which two of its arguments are integers, this rules allows the application of linear constraints $=$, $>$, $<$, $\geq$ and $\leq$ over such arguments. The output data table is the set of tuples for which the chosen constraint is true. For instance, the application of this PR with parameters $\langle (0, 1), < \rangle$ results in the following concept definition:

$$\textit{newC}(A, B) : \textit{integer}(A) \wedge \textit{integer}(B) \wedge \textit{divisors}(A, B) \wedge \textit{less-than}(A, B)$$

which stands for the concept of integers that have divisors greater than the integer they divide. The parameterisation $\langle (0, 1), < \rangle$ compares the arguments in positions 0 and 1 through the relationship $<$.

the embed-algebra and record rules: the embed-algebra PR identifies algebras embedded within concepts in the theory by using axioms provided within the background information. The record PR takes the definition of a function and produces a new function concept whose output is an integer sequence. These two PRs were developed specifically for algebraic domains and details of their implementation are not available.

3. Binary rules: these PRs form a new concept from two old concepts: a primary concept considered as the first input concept to the PR and a secondary concept. To describe the application of the binary PRs the concepts of multiples and square numbers will be used in addition to the concept of divisors. The definitions for these two concepts, supplied in the background domain, are:

$$\textit{multiples}(C, D) : \textit{integer}(C) \wedge \textit{integer}(D) \wedge \textit{multiples}(C, D)$$
$$\textit{non-square}(E) : \textit{integer}(E) \wedge \textit{non-square}(E)$$

where D is a multiple of C, and E is a non-square number.

the compose rule: combines the predicates from two concepts in the new concept

by conjoining the predicates in the new concept definition and removing any duplicates from the output data table. Applying this PR to the concepts of *divisors* and *non-square* numbers, with the parameters $\langle 0, 1\rangle$ yields the following concept definition:

$$\textit{newC}(A, B) : \textit{integer}(A) \wedge \textit{integer}(B) \wedge \textit{divisors}(A, B) \wedge \textit{non-square}(B)$$

which stands for the concept of divisors that are also non-square. The parameters $\langle 0, 1\rangle$ specify that the argument in position 1 of the primary concept should match the argument of the secondary concept.

the disjunct rule: takes two concepts with the same definition and produces the disjunction of their data tables in the new concept. For instance, the application of this PR to the concepts of *divisors* and *multiples* produces the following concept definition:

$$\textit{newC}(A, B) : \textit{integer}(A) \wedge ((\textit{integer}(B) \wedge \textit{divisors}(A, B)) \vee (\textit{integer}(B) \wedge \textit{multiples}(A, B)))$$

which produces the concept of multiples and divisors of an integer. Note that the application of this PR does not require a parameterisation since the definition of both concepts must be the same.

the negate rule: takes two concepts and finds the tuples within the data table of the first concept that complement the data table of the second concept. Applying this PR to the concepts of *non-square* and *integer* results in the following concept definition:

$$\textit{newC}(A) : \textit{integer}(A) \wedge \neg \textit{non-square}(A)$$

which produces the concept of integers that are not non-square numbers. The application of this PR does not require a parameterisation.

the forall rule: takes two concepts whose definitions coincide and generates a new concept containing a universally quantified implication involving the two input data tables. The application of this PR to concepts *divisors* and *multiples* produces the following concept definition:

$$\textit{newC}(A) : \textit{integer}(A) \wedge \forall\, B((\textit{integer}(B) \wedge \textit{divisors}(A, B)) \Rightarrow (\textit{multiples}(A, B)))$$

which stands for a new concept in which if an integer is a divisor of a number implies that the integer is also a multiple of the number. Different parameterisations can be applied through the application of this PR; however, there is not

a description available of how this mechanism works for this PR. Therefore, an explanation of this cannot be provide in this thesis.

the arithmetic rule: performs arithmetic operations (+, -, $*$, div) on specified entries of two input concepts. The output data table contains the result of the chosen arithmetic operation. Assume the concepts C_1 and C_2, which represent the number of divisors and the number of multiples of each integer, have been formed previously by HR as shown below:

$$C_1(A, B) : \textit{integer}(A) \wedge \textit{integer}(B) \wedge B = |\{C : \textit{integer}(C) \wedge \textit{divisors}(A, C)\}|$$
$$C_2(A, B) : \textit{integer}(A) \wedge \textit{integer}(B) \wedge B = |\{C : \textit{integer}(C) \wedge \textit{multiples}(A, C)\}|$$

the application of the arithmetic PR over concepts C_1 and C_2 with parameters $\langle + \rangle$ produces the following concept definition:

$$\textit{newC}(A, B) : \textit{integer}(A) \wedge \exists\, C, D(\textit{integer}(C) \wedge C = |\{E : \textit{integer}(E) \wedge \textit{divisors}(A, E)\}| \\ \wedge \textit{integer}(D) \wedge D = |\{F : \textit{integer}(F) \wedge \textit{multiples}(A, F)\}| \wedge B = C + D)$$

which stands for the concept that adds up the number of divisors and number of multiples of an integer. Note that in this example the parameterisation specifies the use of the addition operation.

the numrelation rule: performs arithmetic comparisons ($<$, $>$, $\leq$, $\geq$) on specified entries of two concepts. The output data table contains the tuples of the original data tables that meet the conditions specified by the comparison operator. Again, applying this PR to concepts C_1 and C_2 with parameters $\langle \geq \rangle$ produces the following concept definition:

$$\textit{newC}(A) : \textit{integer}(A) \wedge \exists\, B, C(\textit{integer}(B) \wedge B = |\{D : \textit{integer}(D) \wedge \textit{divisors}(A, D)\}| \\ \wedge \textit{integer}(C) \wedge C = |\{E : \textit{integer}(E) \wedge \textit{multiples}(A, E)\}| \wedge B \geq C)$$

This concepts defines the integers for which the number of divisors is greater or equal than the number of multiples. Observe that the paremeterisation defines the use of the relationship $\geq$.

In order to fully illustrate the application of the PRs, we show how the concept of prime numbers can be generated via HR. First, the size PR is applied to the concept of divisors (shown in Figure 2.7) with the parameterisation $< 1 >$. This means that the number of tuples for each entry in column 1 of the concept data table is counted, and this number is then recorded for each entry. For instance, in the data table representing the concept of divisors, 1 appears only once in the first column, 2 and 3 appear twice each, and 10 appears four times.

This number is recorded next to the entries in the concept data table as shown in Figure 2.8.

divisors(A,B)	
integer	divisor
1	1
2	1
2	2
3	1
3	3
.	.
.	.
10	1
10	2
10	5
10	10

size < 1 >
⟶

numberOfDivisors(A,B)	
integer	number of divisors
1	1
2	2
3	2
4	3
5	2
6	4
7	2
8	4
9	3
10	4

Definition:
$numberOfDivisors(A, B) = integer(A) \wedge integer(B) \wedge B = |C : integer(C) \wedge divisors(A, C)|$

Figure 2.8: Generation of the concept of number of divisors.

Then, as illustrated in Figure 2.9, the split PR is applied to the new concept with the parameterisation $< 2 = 2 >$. This application of the split PR produces a new data table consisting of those entries in the previous data table whose value in the second column is equals to 2. Through manual inspection of the new concept it is observed that it represents the concept of prime numbers.

numberOfDivisors(A,B)	
integer	number of divisors
1	1
2	2
3	2
4	3
5	2
6	4
7	2
8	4
9	3
10	4

split< 2 = 2 >
⟶

prime(A)
2
3
5
7

Definition:
$prime(A) = integer(A) \wedge integer(2) \wedge 2 = |B : integer(B) \wedge divisors(A, B)|$

Figure 2.9: Generation of the concept of prime numbers.

2.3.1.3 Conjecture making

Each time a new concept is generated, HR checks to see whether it can make conjectures with it. This could be equivalence conjectures, if the new concept has the same data table as a previous concept; implication conjectures, if the data table of the new concept either

subsumes or is subsumed by that of another concept, or non-existence conjectures, if the data table for the new concept is empty.

Following the running example of prime numbers, we show how HR identifies the conjecture that all prime numbers are non-squares. After the concept of prime numbers has been formed, HR checks to see whether the data table is equivalent to, subsumed by, or subsumes another data table, or whether it is empty. Assuming the concept of non-square numbers has been previously generated by HR, the data tables of both the concept of prime numbers and the concept of non-square numbers, shown in Figure 2.10, are compared. Note that in this case we assume the concept of non-square numbers was developed by HR instead of given in the background domain. Details of how this concept is invented by HR can be found in [104, Chapter 3] where it is shown the steps to generate this concept through the application of the *match*, *exists* and *negate* PRs.

prime(A)
2
3
5
7

$\Rightarrow$

non-square(A)
2
3
5
6
7
8
10

Definitions:
$prime(A) = integer(A) \wedge integer(2) \wedge 2 = |B : integer(B) \wedge divisors(A, B)|$
$non\text{-}square(A) = integer(A) \wedge \neg \exists B(integer(B) \wedge divisors(A, B) \wedge A = B * B)$

Figure 2.10: Concepts representing the prime and non-square numbers between 1 and 10.

HR would immediately see that all of its prime numbers are also non-squares, and so conjectures that this is true for all prime numbers. That is, it will make the following implication conjecture:

$$prime(A) \Rightarrow non\text{-}square(A)$$

which in HR output is given as:

$$\forall A(integer(A) \wedge integer(2) \wedge 2 = |B : integer(B) \wedge divisors(A, B)| \Rightarrow integer(A) \wedge \neg \exists C(integer(C) \wedge divisors(A, C) \wedge A = C * C)$$

This illustrates the process of theory formation within HR. For a more detailed description about HR and its capabilities the reader is referred to [37, 32].

2.3.1.4 Challenges in using HR in the formal modelling context

As mentioned earlier in this section ATF and in particular HR has been successfully applied in different domains; however, at the moment of writing this thesis there is no previous account of using HR in the context of formal modelling. In general, using HR to form theories about formal models and identifying interesting conjectures as candidate invariants presents with some challenges:

1. HR produces a large number of conjectures – in our experiments some were in the range of 3000 to 12000 conjectures per run – from which only a very small set represent interesting conjectures. Our main challenge was to find a way of automatically selecting the conjectures that are interesting for the domain among the conjectures obtained from HR; in this way, only a handful of candidate invariants is presented to the user instead of a large set of uninteresting conjectures.

2. The HR theory formation mechanism consists of an iterative application of production rules over all concepts in the theory. In order for HR to perform an exhaustive search, all possible combinations of production rules and concepts must be carried out. However, there is not a fixed number of theory formation steps set up for this process, since this varies depending on the domain, i.e. some domains need more theory formation steps than others. This represented a challenge for the use of HR in the discovery of invariants since it was possible that an invariant had not been formed only because not enough steps had been applied, and performing an exhaustive search would give rise to an unmanageable number of conjectures.

3. Some production rules are more effective in certain domains than others. Selecting the appropriate production rules results in the construction of a more interesting theory. For instance, if we are looking at a refinement step in an Event-B model that introduces a partition of sets, we expect the new invariants to define properties over the new sets; therefore, production rules such as the *arithmetic PR* will not be of much interest in the development of the theory associated to the refinement step. Automatically selecting appropriate production rules requires knowledge about the domain.

4. HR production rules were created for the context of mathematical domains, this means that some type of invariants cannot be generated by HR. For instance, HR cannot invent invariants that contain recursive definitions or invariants that require the inverse type of a concept. In Section 4.2.1.1 a possible interesting new production rule is identified.

5. HR does not support sets as parameter to a concept; therefore, common Event-B expressions, such as sets of sets cannot be represented in HR. The parameters of a concept

are treated as strings; thus, the operations performed by the PRs are based on string comparisons. In order to add support for sets we need to add support for strings that represent lists of elements and add list handling in the required PRs.

Challenges 4 and 5 require modifying the core of HR which is not in the scope of this thesis. However, future directions of work are described in Section 8.3. In addressing challenges 1, 2 and 3, a set of heuristics were developed which automatically constrain both the configuration of HR and the selection of conjectures that arise from a theory formation run. In the following chapter we describe how ATF can be applied to form theories of Event-B models as well as present the heuristics and their application in detail.

2.4 Summary

Formal methods are supported by a wide range of tools and techniques. As shown in this chapter, these techniques can be applied at different stages of the development cycle. Moreover, different aspects of the development can be targeted by formal approaches; for instance, refinement and automated invariant discovery aid formal modelling, while theorem proving and model checking aid formal verification.

We have described in detailed Event-B, a formalism for the development of discrete event systems that provides support for a refinement-style of modelling. The Rodin platform has also been outlined, an eclipse-based tool used for the development of Event-B models. In particular, we have described ProB, a Rodin plug-in for the simulation and model checking of Event-B models. Finally, ATF was presented, a machine learning technique whose purpose is to form theories about objects of interest in a domain. In particular, we have outlined HR, a system that implements ATF. HR attempts to construct new concepts through the use of a set of production rules and to formulate conjectures that relate those concepts.

The work presented in this thesis focuses on formal methods tailored for the design stage. Specifically, our techniques aim at providing modelling guidance for refinement-based formal modelling. For this purpose we built upon Event-B; specifically, ProB is exploited, alongside HR, for the dynamic analysis of simulation traces in order to discover invariants of Event-B models. Furthermore, proof failure analysis and patterns of refinement are key aspects to guide the search for invariants as well as to provide modelling guidance when a step fails but is close to a known pattern of refinement.

Chapter 3

Invariant Discovery through HR

The posit and prove style of refinement adopted by Event-B gives the developer the freedom of incrementally introducing design details; however, the user is responsible for ensuring the correctness of each refinement step by proving the generated POs. Discharging such POs typically requires a developer to supply properties – properties that relate to their design decisions. This chapter presents a heuristic approach that supports the activity of formal modelling by automatically discovering such properties through the use of ATF, simulation and proof.

3.1 Approach

The approach consists of exploring simulation traces of formal models in order to identify properties that are true over such traces. There are three main components involved:

1. a formal modelling component that supports proof;
2. a simulation component that generates system traces; and
3. an ATF component that generates conjectures from the analysis of the traces.

Figure 3.1 illustrates how these three components interact with each other.

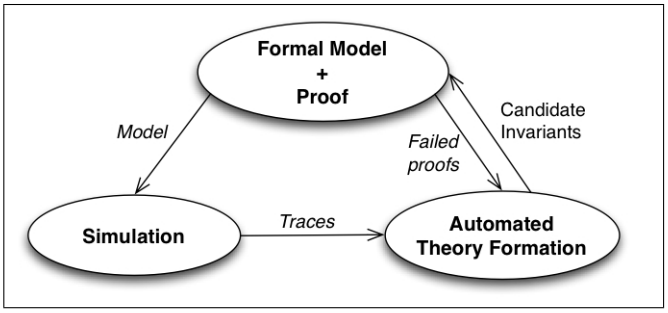


Figure 3.1: Approach for the automatic discovery of invariants.

The formal model is simulated in order to generate traces that specify the state of the system in different scenarios; these traces along with proof failure analysis are used to tailor the theory formation process. The result from this is a set of candidate invariants which are fed back to the model and presented to the user, who makes the final decision about which invariants should be introduced into the model. From a modelling perspective we have focused on Event-B and the Rodin tool-set, in particular we have used the ProB animator plug-in [86] for the simulation component. In terms of ATF, we use the HR system.

3.1.1 Construction of conjectures within the Event-B formalism

The model in Figure 3.2 is used to show how a gluing invariant can be generated through the use of theory formation. At the abstract level the boolean variable *full* is modified in event *addA* through a non-deterministic substitution when *full* is false – note that the non-deterministic substitution, i.e. *full* :∈ *BOOL*, specifies that variable *full* is non-deterministically assigned a boolean value. At the concrete level the state of the system is refined by replacing the abstract variable *full* by concrete variables *x* and *m*. Moreover, the abstract event *addA* is refined by the concrete event *addC*, which gradually increments variable *x* by one unit when *x* is less than *m*.

ABSTRACT LEVEL:	**CONCRETE LEVEL:**
Variables	**Variables**
full	x m
Invariants	**Invariants**
full ∈ BOOL	x ∈ ℕ
	m ∈ ℕ
Events	**Events**
Initialisation	**Initialisation**
then	**then**
full := false	x := 0
end	m := 3
Event addA ≙	**end**
when	**Event** addC ≙
full = false	**refines** addA
then	**when**
full :∈ BOOL	x < m
end	**then**
	x := x+1
	end

Figure 3.2: Flawed Event-B model.

As it stands the model generates a failed guard strengthening PO, shown in (3.1). This

PO specifies that the concrete guard $x<m$ must imply the abstract guard $full=false$.

$$x < m \vdash full = false \tag{3.1}$$

The failure is generated because the relationship between the abstract and concrete state; i.e. the gluing invariant, has not been defined.

The first step in the discovery of invariants is to simulate the Event-B model. The aim of the simulation is to obtain the set of examples required by HR to invent conjectures about a domain. The animation trace shown in Figure 3.3 is produced by the ProB simulator for the flawed Event-B model. The trace shows the value of the abstract and concrete variables at each step of the simulation.

		Animation steps			
	Variables	**S1**	**S2**	**S3**	**S4**
Abstract	full	false	false	false	true
Concrete	x	0	1	2	3
	m	3	3	3	3

Figure 3.3: Animation trace generated by the ProB simulator.

The input background information required by HR is extracted from the Event-B model and the simulation trace. This consists of the set of core concepts that describe the domain; i.e. the data types and the state of the model, with their respective examples. Figure 3.4 shows the data tables of the core concepts corresponding to the model. That is, the concept *state* which represents every step of the animation trace, the concepts *boolean* and *integer* which are the data types and and the concepts *full, x* and *m* which are the abstract and concrete variables of the model.

state(A)
S1
S2
S3
S4

boolean(B)
true
false

integer(C)
0
1
2
3

full(A,B)	
S1	false
S2	false
S3	false
S4	true

x(A,C)	
S1	0
S2	1
S3	2
S4	3

m(A,C)	
S1	3
S2	3
S3	3
S4	3

Figure 3.4: Core concepts.

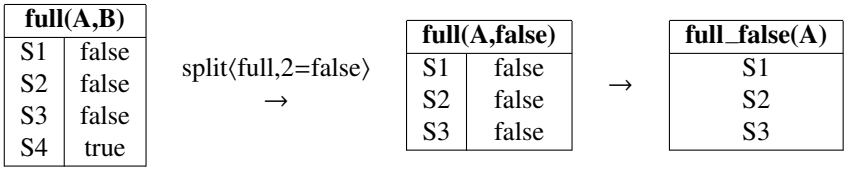
With this background information HR applies all possible combinations of concepts and production rules in order to generate new concepts and form conjectures. After 1000 theory formation steps HR generates a total of 2083 conjectures. Through manual inspection, conjecture (3.2) is identified as the missing invariant of the flawed model – the equivalent Event-B representation of the conjecture is shown in (3.3). Note that this transformation is

carried out manually by the user.

$$\forall A(state(A) \wedge boolean(FALSE) \wedge full(A, FALSE) \Leftrightarrow$$
$$state(A) \wedge \exists B, C(integer(B) \wedge x(A, B) \wedge integer(C) \wedge m(A, C) \wedge B < C)) \quad (3.2)$$
$$full = false \Leftrightarrow x < m \quad (3.3)$$

This conjecture is built by HR in three steps:

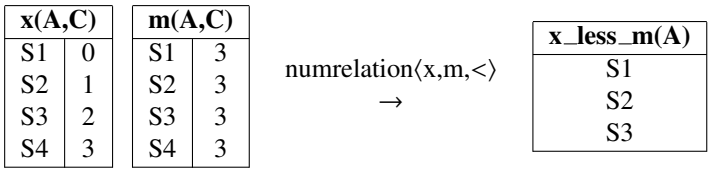
1. HR produces the concept of the set of states for which *full* is equals to *false*. This concept is generated through the application of the *split* production rule as illustrated in Figure 3.5. As can be observed an intermediate output datatable is generated with all tuples of concept *full* whose second column is equals to *false*. Since the second column is the same for all tuples of the intermediate concept, this column is removed from the final output concept; i.e. *full_false(A)*.



$full\_false(A)$ = $state(A) \wedge boolean(false) \wedge full(A, false)$

Figure 3.5: Generating the concept of states for which full is false.

2. Then HR produces the concept of the set of states for which x is less than m. This concept is generated by HR through the application of the *numrelation* production rule. The application of this step is illustrated in Figure 3.6. The output concept lists all the states for which $x<m$.



$x\_less\_m(A)$ = $state(A) \wedge (\exists B, C(integer(B) \wedge x(a, B) \wedge integer(C) \wedge m(A, C) \wedge B < C))$

Figure 3.6: Generating the concept os states for which x is less than m.

3. Immediately after the generation of new concepts, HR looks for relationships with other existing concepts. As shown in Figure 3.7, HR finds that the concept *full_false(A)* has the same list of examples as concept *x_less_m(A)*, giving rise to the equivalence conjecture (3.2), which represents the missing invariant.

full_false(A)
S1
S2
S3

$\Leftrightarrow$

x_less_m(A)
S1
S2
S3

$$\forall\, A(state(A) \wedge boolean(FALSE) \wedge full(A, FALSE) \Leftrightarrow$$
$$state(A) \wedge \exists\, B, C(integer(B) \wedge x(A, B) \wedge integer(C) \wedge m(A, C) \wedge B < C))$$

Figure 3.7: Identified equivalence conjecture.

In this section it was illustrated how theories of Event-B models can be formed by HR. A simple Event-B model was used to identify a missing invariant within the set of conjectures generated during the theory formation process. In the next section the model of the Mondex system developed by Butler and Yadav [22] is used in order to show how this process is performed automatically.

3.2 Automatic invariant discovery of Event-B models

The Mondex system models a protocol for money transfer that ensures that no money is lost in a transaction regardless of the success or failure of the transaction. The model developed in [22] is composed of one abstract model and eight refinement steps as follows:

Abstract model: models successful and failed transfers of money through atomic steps that modify the balance of the purses involved in the transaction.

Level 2: introduces states for a transaction as well as the concepts of source, target and amount within a transaction.

Level 3: removes some redundant variables from the model.

Level 4: introduces dual states for each side of the transaction.

Level 5: uses messaging between purses instead of allowing direct access to their state information.

Level 6: limits the purses to participate only in one transaction at a time.

Level 7: controls the freshness of a transaction by introducing a history of sequence numbers used by each purse involved in the transaction.

Level 8: replaces the history of sequence numbers with a single counter for each purse that is increased every time the purse is in a new transaction.

Level 9: replaces the sets that model the states of the purses by a function that maps a purse to its status.

Details of each of these steps can be found in [22]. Here the focus is on the last refinement step. A fragment of the model is shown in Figure 3.8.

ABSTRACT LEVEL:

Context:
Sets
purses, trans

Model:
Variables
currentF, currentT, active,
idleFP, eprP, epaP, aborteprP,
abortepaP, endFP, idleTP,
epvP, abortepvP, endTP
Invariants
active $\subseteq$ purses
currentF $\in$ purses $\nrightarrow$ trans
currentT $\in$ purses $\nrightarrow$ trans
partition(active,dom(currentF),dom(currentT))
partition(dom(currentF),idleFP,eprP,epaP,
aborteprP,abortepaP,endFP)
partition(dom(currentT),idleTP,epvP,
abortepvP,endTP)
Events
...
StartFrom $\widehat{=}$
any t, p1
where
p1 $\in$ idleFP
...
then
eprP := eprP $\cup$ $\{p1\}$
idleFP := idleFP $\setminus$ $\{p1\}$
...
end

CONCRETE LEVEL:

Context:
Sets
status
Constants
IDLEF, EPR, EPA, ABORTEPR,
ABORTEPA, ENDF, IDLET, EPV,
ABORTEPV, ENDT
Axioms
partition(status,{IDLEF},{EPR},{EPA},
{ABORTEPR},{ABORTEPA},{ENDF},
{IDLET},{EPV},{ABORTEPV},{ENDT})

Model:
Variables
statusF
Invariants
statusF $\in$ purses $\nrightarrow$ status
Events
...
StartFrom $\widehat{=}$
refines StartFrom
any t, p1
where
p1 $\mapsto$ IDLEF $\in$ statusF
...
then
statusF(p1) := EPR
...
end
...

Figure 3.8: A refinement step from the Mondex [22] development.

The *StartFrom* event handles the initiation of a transaction on the side of the source purse. In order to initiate a transaction, the source purse, i.e. *p1*, must be in the *idle* state (waiting state) and after the transaction has been initiated the state of the purse must be changed to *epr* (expecting request). As shown in Figure 3.8, at the abstract level a purse is *active* if it is the source of a transaction, i.e. it belongs to the domain of function *currentF*, or the target, i.e. it belongs to the domain of function *currentT*. Moreover, the state of a purse is represented by disjoint sets, e.g. the variables *eprP* and *idleFP*. At the concrete level the representation

is changed to a function, i.e. the variable *statusF*, a mapping from the set of purses to an enumerated set (state), e.g. the constants *IDLEF* and *EPR*.

Note that the refinement presented in Figure 3.8 is unprovable as it stands since the relationship between the abstract and concrete states has not been defined. In order to verify the refinement a gluing invariant is required. That is:

$$\text{idleFP} = \text{statusF}^{-1}[\{\text{IDLEF}\}] \tag{3.4}$$

This invariant states that the abstract set *idleFP* can be obtained from the inverse of the function *statusF* evaluated over the enumerated set *IDLEF* – a similar gluing invariant would be required for the abstract set *eprP* and the function *statusF*. Next, it is shown how (3.4) is generated via HR. Furthermore, we motivate the need for the development of some heuristics in order to optimise the theory formation process and to automate the selection of candidate invariants.

Figure 3.9 describes the flow of data for the invariant discovery process. Concepts are extracted directly from an Event-B model, while examples are derived from simulation traces. The result of the theory formation process is a set of conjectures from which some may represent candidate invariants of the given Event-B model. Details of how this data is obtained are presented next. We use the refinement step of the Mondex model as a running example.

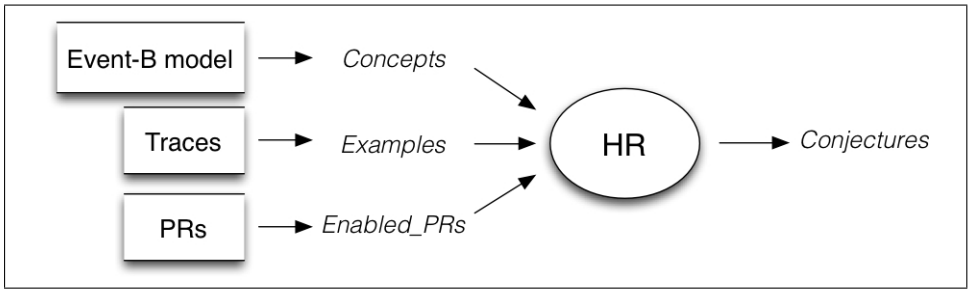


Figure 3.9: Discovery of invariants in Event-B models through HR.

3.2.1 Extracting concepts

There are three types of concepts in HR:

T1: user-given concepts that enumerate the objects of interest,

T2: user-given concepts that define features of the objects of interest, and

T3: concepts automatically generated by HR through the use of the PRs.

Within Event-B, the input concepts are defined by the state of the model which is represented via *carrier sets*, *constants* and *variables*. Carrier sets are considered to be T1 concepts because they represent the basic entities in an Event-B model, i.e. they are user defined types

from which constants and variables can be formed. Variables are considered to be T2 concepts since they are defined in terms of the objects of interest, i.e. the carrier sets. Constants also represent T2 concepts unless they are members of a carrier set, in which case they are considered to be examples of the carrier set. More formally, the sets of T1 and T2 concepts are defined as follows:

$$conceptsT1(M) \triangleq carrierSets(M) \tag{3.5}$$

$$conceptsT2(M) \triangleq variables(M) \cup \{c \mid c \in constants(M) \wedge c \notin \bigcup carrierSets(M)\} \tag{3.6}$$

where *carrierSets(M)*, *variables(M)* and *constants(M)* denote carrier sets, variables and constants associated with model M respectively.

To illustrate the process of concept extraction, consider again the refinement step from our running example. Figure 3.10 summarises the abstract and concrete state associated with this refinement step.

Abstract Level:	**Concrete Level:**
Sets:	*Sets:*
purses, trans	status
Variables:	*Constants:*
active, currentF, currentT, idleFP, eprP, epaP, aborteprP, abortepaP, endFP, idleTP, epvP, abortepvP, endTP	IDLEF, EPR, EPA, ABORTEPR, ABORTEPA, ENDF, IDLET, EPV, ABORTEPV, ENDT
	Variables:
	statusF

Figure 3.10: State in the abstract and concrete levels of the Mondex refinement step.

The results of applying functions (3.5) and (3.6) to the abstract and concrete models is shown in Figure 3.11.

$conceptsT1$(*abstractModel*) = {purses, trans}
$conceptsT2$(*abstractModel*) = {active, currentF, currentT, idleFP, eprP, epaP, endFP, idleTP, epvP, endTP, aborteprP, abortepaP, abortepvP}

(a) Core concepts obtained from the abstract model.

$conceptsT1$(*concreteModel*) = {status}
$conceptsT2$(*concreteModel*) = {statusF}

(b) Core concepts obtained from the concrete model.

Figure 3.11: Core concepts extracted from the Mondex refinement step.

Note that the constants associated with the concrete level, i.e. $IDLEF$, EPR, etc, are not identified as core concepts because they are subsets of the set *status*, as defined by the axiom:

$$\begin{aligned} &\mathit{partition}(\mathit{status}, \{\mathit{IDLEF}\}, \{\mathit{EPR}\}, \{\mathit{EPA}\}, \{\mathit{ABORTEPR}\}, \{\mathit{ABORTEPA}\}, \\ &\qquad \{\mathit{ENDF}\}, \{\mathit{IDLET}\}, \{\mathit{EPV}\}, \{\mathit{ABORTEPV}\}, \{\mathit{ENDT}\}) \end{aligned}$$

The next step involves translating the core concepts into HR definitions. An HR definition consists of a functor along with the type of its parameters. The required type information is given as invariants and axioms within a model. In the case of the Mondex example, the invariants shown in Figure 3.12 specify the types of the T2 concepts listed in Figure 3.11. From these invariants it is observed that the abstract variables *currentF* and *currentT* are function from *purses* to *trans*, variables *active, idleFP, eprP, etc.* are subsets of *purses*, and the concrete variable *statusF* is a function from *purses* to *status*. In terms of a HR definition, *idleFP* is represented as *idleFP(A)* where *A* denotes the type of *purses*. Similarly, *statusF* is represented as *statusF(A, B)* and *currentF* is represented as currentF(A, C), where *A, B* and *C* denote the types *purses, status* and *trans*, respectively. Figure 3.13 shows the HR definitions for all the core concepts that arise in the running example.

Abstract invariants:
active $\subseteq$ purses
currentF $\in$ purses $\nrightarrow$ trans
currentT $\in$ purses $\nrightarrow$ trans
partition(active,dom(currentF),dom(currentT))
partition(dom(currentF),idleFP,eprP,epaP, aborteprP,abortepaP,endFP)
partition(dom(currentT),idleTP,epvP, abortepvP,endTP)

Concrete invariant:
statusF $\in$ purses $\nrightarrow$ status

Figure 3.12: Core concepts type invariants.

Types:	**Definitions:**			
A $\equiv$ purses	active (A)	epaP(A)	eprP(A)	epvP(A)
B $\equiv$ status	endFP(A)	endTP(A)	idleFP(A)	idleTP(A)
C $\equiv$ trans	abortepaP(A)	aborteprP(A)	abortepvP(A)	statusF(A,B)
	currentF(A,C)	currentT(A,C)		

Figure 3.13: Definitions of the Mondex core concepts.

Details about how the data tables of these concepts are built will be given in the next sections. Note that the process of identifying the core concepts for a given Event-B refinement step is fully automatic. Details on the automated process are presented in Chapter 4.

3.2.2 Generating examples of concepts

For each concept definition, HR requires a set of examples in order to apply its PRs. As mentioned earlier, within the context of Event-B, simulation provides a source of such examples. Through simulation it is possible to analyse the operation of an Event-B model by observing how its state changes when different scenarios are explored. We use the ProB animator in order to construct *simulation traces* of Event-B models, from which the required examples are extracted.

A trace represents a record of the behaviour of the system during the simulation, i.e. it contains the value of the domain data at each step. Drawing upon the Mondex refinement step, Figure 3.14 shows a fragment of a trace generated from a simulation run performed by ProB.

Sets and Constants (*all states*)	
purses	**status**
purses1,purses2,purses3, purses4,purses5	IDLEF,EPR,EPA,ABORTEPR,ABORTEPA, ENDF,IDLET,EPV,ABORTEPV,ENDT

	Variables	
state	**idleFP**	**statusF**
S0	-	-
S1	-	-
S2	-	-
S3	-	-
S4	-	-
S5	purses3	purses3↦IDLEF
S6	purses3, purses5	purses3↦IDLEF,purses5↦IDLEF
S7	purses5	purses3↦EPR,purses5↦IDLEF
S8	purses5	purses3↦EPR,purses5↦IDLEF
S9	-	purses3↦EPR,purses5↦EPR
S10	-	purses3↦EPR,purses5↦EPA
⋮	⋮	⋮
S58	-	purses1↦ABORTEPA,purses5↦ENDF
S59	-	purses1↦ABORTEPA

Figure 3.14: An example animation trace.

Note that we have only shown the concepts that are needed for the generation of the gluing invariant (3.4). Note also that carrier set *purses* and variable *idleFP* denote abstract

concepts, while the carrier set *status* and variable *statusF* denote concrete concepts. In order to observe gluing invariants we must be able to link the abstract and concrete states. To achieve this, *state* is added as a core concept to the domain of an Event-B model – where a state represents a step in the simulation trace .

In terms of generating examples, when the axioms of an Event-B model enumerate the members of a carrier set, ProB assigns these as the examples of the carrier set. For instance, ProB automatically selects the constants *IDLEF, EPR, EPA,* etc. as the examples of *status*. In the case of carrier sets such as *purses*, where there are no axioms defining membership, ProB dynamically generates a list of arbitrary members using the name of the carrier set as a prefix. To illustrate, for *purses* ProB generates the list *purses1, purses2, purses3, purses4* and *purses5*. Note that once set at the beginning of the simulation, the values of carrier sets and constants remain the same in all steps of the simulation trace.

The examples of variables and the state concept vary per step within the simulation trace. Each step represents the execution of an event; therefore, only the value of the variables modified by the event are changed in each step. It is also possible that a variable does not have a value associated with a state; for instance, a set that is empty at a particular moment in the simulation. This is represented in Figure 3.14 with the symbol '–'. The examples of the state concept consists of an ID list that indicates the sequence of steps in the trace. The ID of a state is represented by the letter *S* attached with a sequence number. Each step within the simulation trace represents the execution of an event. While the events themselves are not part of the trace since the invariants specify global properties of the data, their effect on the value of the variables is recorded. The examples of the state concept are obtained by enumerating each step within the trace.

Simulations can be generated randomly or through the use of test case generators. The quality of the invariants depends on the quality of the simulations, so the use of test case generators is highly recommended whenever possible. For the Rodin toolset, no test case generator is available. The simulations used in our experiments are randomly generated as the development of a test case generator is outside of the scope of this thesis. We come back to this in Chapter 8.

3.2.3 Constructing data tables

HR does not work directly with simulation traces; it works instead with *data tables*. For each concept, a data table is constructed as follows:

$$DataTable(x) \triangleq \text{if } isa\_conceptT1(x) \text{ then } \{e \mid e \in x\} \\ \text{else } \{< s, e > \mid s \in state \ \wedge \ e \in trace(s, x)\} \tag{3.7}$$

where $isa\_conceptT1(x)$ is true iff x denotes a T1 concept, and *trace*(s, x) computes the set of examples e for concept x in state s of a previously generated simulation trace. Note that in the case of a T1 concept, the associated data table simply enumerates the elements of the concept. For example, applying (3.7) to the T1 concepts given in Figure 3.14 yields:

$$DataTable(purses) = \{purses1, purses2, purses3, purses4, purses5\}$$
$$DataTable(status) = \{IDLEF, EPR, EPA, ABORTEPR, ..., ENDT\}$$

The corresponding data tables are given in Figure 3.15. Note that a data table for *state* is always required in order to link abstract and concrete perspectives. In the case of a T2 concept, the data tables corresponds to a set of tuples that associate examples of the concept with the states in which they appear within the trace. To illustrate, the application of (3.7) to the T2 concepts given in Figure 3.14 yields:

$$DataTable(idleFP) = \{< S5, purses3 >, < S6, purses3 >, ..., < S54, purses4 >\}$$
$$DataTable(statusF) = \{< S5, purses3 \mapsto IDLEF >, ..., < S59, purses1 \mapsto ABORTEPA >\}$$

The corresponding data tables are shown in Figure 3.16.

purses
purses1
purses2
purses3
purses4
purses5

status
IDLEF
EPR
EPA
ABORTEPR
ABORTEPA
ENDF
IDLET
EPV
ABORTEPV
ENDT

state
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
⋮
S58
S59

Figure 3.15: HR data tables type 1 core concepts.

3.2.4 Selecting PRs and running HR

Once provided with data tables, HR applies all possible combinations of concepts and PRs in order to generate new concepts and form conjectures. HR currently contains a set of 22 PRs. In our work we have focused on 7, which we found relevant to the Event-B formalism.

idleFP(A,B)	
S5	purses3
S6	purses3
S6	purses5
S7	purses5
S8	purses5
S19	purses4
S25	purses5
S29	purses1
S30	purses1
S31	purses1
S38	purses5
S39	purses5
S40	purses5
S44	purses5
S45	purses5
S52	purses5
S53	purses5
S54	purses5

statusF(A,B,C)		
S5	purses3	IDLEF
S6	purses3	IDLEF
S6	purses5	IDLEF
S7	purses3	EPR
S7	purses5	IDLEF
S8	purses3	EPR
S8	purses5	IDLEF
⋮	⋮	⋮
S29	purses1	IDLEF
S29	purses5	ABORTEPR
S30	purses1	IDLEF
S30	purses5	ABORTEPR
S31	purses1	IDLEF
S31	purses5	ABORTEPR
⋮	⋮	⋮
S59	purses1	ABORTEPA

Figure 3.16: Example HR data tables for type 2 core concepts.

Table 3.1 shows the correspondence between the 7 PRs and the Event-B operators. However, as will be explained in Chapter 4, we believe that there is scope for new PRs that address aspects of the formalism not currently covered.

Production Rule	**Formula Operator**
negate	$\neg$
compose	$\wedge$, $\cap$, $\lhd$, $\not\lhd$, $\rhd$, $\not\rhd$
disjunct	$\vee$, $\cup$
arithmetic	$+$, $-$, $\times$, div
numrelation	$>$, $\geq$, $<$, $\leq$, $=$
exists	dom, ran
split	when a member of a finite set is identified.

Table 3.1: Compatibility between production rules and formula operators.

In order to illustrate HR's theory formation mechanism, consider again the data tables given in Figures 3.15 and 3.16. Starting with these tables and all the PRs mentioned in Table 3.1 enabled, HR is run for 1000 theory formation steps. Through our experience most interesting conjectures have been formed within 1000 steps. Moreover, it has been noted that with the Event-B domain models, the theory formation process slows down not far after these 1000 steps. Thus our selection of length. After 433 steps of the theory formation run,

HR forms the following conjecture:

$$\forall A, B.(\textit{state}(A) \wedge \textit{purses}(B) \wedge \textit{idleFP}(A, B) \Leftrightarrow \textit{status}(\textit{IDLEF}) \wedge \textit{statusF}(A, B, \textit{IDLEF})) \quad (3.8)$$

This conjecture specifies that the concept *IdleFP(A,B)* is equivalent to the concept *statusF(A,B,C)* when *C* is instantiated to be *IDLEF*. In the context of the Mondex refinement step, this means that the set *idleFP* (abstract) is equal to the inverse image of the singleton set {*IDLEF*} under the function *statusF* (concrete), i.e.

$$\text{idleFP} = \text{statusF}^{-1}[\{\text{IDLEF}\}]$$

which is gluing invariant (3.4) – note that the transformation of (3.8) to its Event-B representation is not currently automatic. This is addressed as future work in Section 8.3.

The discovery of this conjecture involves noticing that data table **idleFP(A,B)** is identical to the rows of data table **statusF(A,B,C)** for which **C** is equals to **IDLEF**. HR's *split* PR plays a central role in such discovery as illustrated in Figure 3.17. Within the theory formation process, when the *split* PR is applied with the parameters <3, IDLEF>, a specialisation of concept *statusF* is formed, i.e. a data table is formed by extracting the tuples of concept *statusF* whose third column matches the parameter *IDLEF*. Note that since the third column is the same for all tuples, this column is removed from the output concept. Immediately after the generation of this concept, i.e. *status_IDLEF*, HR discovers that its datatable has the same list of examples as concept *idleFP*, giving rise to conjecture (3.8).

Input		
statusF(A,B,C)		
S5	purses3	IDLEF
S6	purses3	IDLEF
S6	purses5	IDLEF
S7	purses3	EPR
S7	purses5	IDLEF
S8	purses3	EPR
S8	purses5	IDLEF
⋮	⋮	⋮
S29	purses1	IDLEF
S29	purses5	ABORTEPR
S30	purses1	IDLEF
S30	purses5	ABORTEPR
S31	purses1	IDLEF
S31	purses5	ABORTEPR
⋮	⋮	⋮
S59	purses1	ABORTEPA

split<3,IDLEF> →

Intermediate		
statusF(A,B,IDLEF)		
S5	purses3	IDLEF
S6	purses3	IDLEF
S6	purses5	IDLEF
S7	purses5	IDLEF
S8	purses5	IDLEF
S19	purses4	IDLEF
S25	purses5	IDLEF
S29	purses1	IDLEF
S30	purses1	IDLEF
S31	purses1	IDLEF
S38	purses5	IDLEF
S39	purses5	IDLEF
S40	purses5	IDLEF
S44	purses5	IDLEF
S45	purses5	IDLEF
S52	purses5	IDLEF
S53	purses5	IDLEF
S54	purses5	IDLEF

→

Output	
statusF_IDLEF(A,B)	
S5	purses3
S6	purses3
S6	purses5
S7	purses5
S8	purses5
S19	purses4
S25	purses5
S29	purses1
S30	purses1
S31	purses1
S38	purses5
S39	purses5
S40	purses5
S44	purses5
S45	purses5
S52	purses5
S53	purses5
S54	purses5

Given a data table T, *split<M, N>* derives a new data table T' such that all the rows in T' are identical to the rows in T where the column M has value N.

Figure 3.17: Concept of purses whose status is IDLEF.

statusF_IDLEF(A,B)	
S5	purses3
S6	purses3
S6	purses5
S7	purses5
S8	purses5
S19	purses4
S25	purses5
S29	purses1
S30	purses1
S31	purses1
S38	purses5
S39	purses5
S40	purses5
S44	purses5
S45	purses5
S52	purses5
S53	purses5
S54	purses5

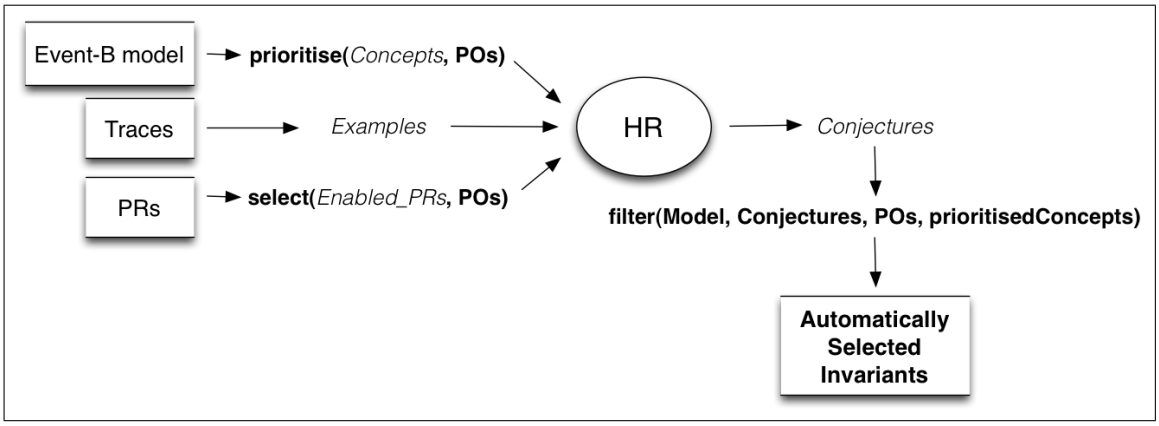
$\Leftrightarrow$

idleFP(A,B)	
S5	purses3
S6	purses3
S6	purses5
S7	purses5
S8	purses5
S19	purses4
S25	purses5
S29	purses1
S30	purses1
S31	purses1
S38	purses5
S39	purses5
S40	purses5
S44	purses5
S45	purses5
S52	purses5
S53	purses5
S54	purses5

$$\forall A, B.(state(A) \land purses(B) \land status(IDLEF) \land statusF(A, B, IDLEF) \Leftrightarrow idleFP(A, B))$$

Figure 3.18: Formed equivalence conjecture.

3.3 Heuristic Approach



prioritise(*Concepts*, POs): concepts that occur within failed POs are prioritised during theory formation.

select(*Enabled_PRs*, POs): the selection of the enabled PRs is determined by the relationships specified between the concepts that occur within failed POs.

filter(Model, Conjectures, POs, prioritisedConcepts): the filtering of the conjectures (candidate invariants) is driven in part by the concept prioritisation.

Figure 3.19: Approach for the automatic discovery of invariants.

As illustrated above, the use of HR in discovering Event-B invariants can involve significant user interaction. In particular, the user must supply domain information about their

models, the set of PRs that are to be enabled, as well as analyse the large number of conjectures formed during theory formation. For instance, recall in Section 3.2.4 where invariant (3.4) was discovered by an application of the split PR to the concept *statusF* using the parameter <3, IDLEF>. Without guidance, HR's search for such parameter settings makes invariant discovery infeasible. As particular combinations of these parameters turn out to be useful for different domains, finding the right combination is largely a process of trial and error. To achieve such guidance, we have developed a set of heuristics. Figure 3.19 provides a high-level picture of how these heuristics relate to the basic invariant discovery diagram given in Figure 3.9.

Our heuristics exploit the strong interplay between modelling and reasoning in Event-B. Specifically, central to the design of our heuristics is the observation that proof-failure analysis provides insights into the structure of missing invariants. Figure 3.19 illustrates this by showing that each heuristic is parametrised by the POs associated with a failed refinement step; which implies that the invariant discovery search is guided by the failed POs. The heuristics are used both in configuring HR as well as filtering the conjectures. In other words, we required heuristics to automatically focus the theory formation to each Event-B domain defined by a user. The heuristics achieve this through the syntactic analysis of the model and the POs, which allows the *prioritisation* in the conjecture generation and the *filtering* of the theory formation output. Below we provide definitions and explanations for each heuristic, focusing first on the *Configuration Heuristics* (CH) in Section 3.3.1 and then the *Filtering Heuristics* (FH) in Section 3.3.2. The example shown in Section 3.1.1 is used to illustrate the application of the heuristics.

3.3.1 Configuration heuristics

Every time HR forms a new concept it tries all applicable theory formation steps; i.e. all applicable combinations of PRs and concepts that involve the new concept. Furthermore, most PRs can be applied with different parameterisations. This means that the theory formation process can lead to a combinatorial explosion [34]. HR uses an *agenda* mechanism to organise how concepts are explored. We use the CH heuristics to influence this agenda mechanism and to constrain the applicable PRs during theory formation. This is achieved by prioritising the concepts of interest according to the failed POs and by choosing only PRs that can rise interesting relationships between those concepts. We use two overall heuristics, i.e. CH1 and CH2, when configuring HR for a given Event-B refinement step:

CH1. *Prioritise core and non-core concepts that occur within the failed POs.*

Through this heuristic a higher priority is given to core and non-core concepts that occur within the failed POs during theory formation. Consequently, the prioritised concepts are placed in the top of HR's agenda which results in the earlier generation

of conjectures that relate to these concepts. The set of core and non-core concepts associated with the set of failed POs is organised as follows:

$$
\begin{aligned}
&prioritise(Concepts, POs) \triangleq \\
&\quad \{pri(c,1) \mid c \in \textit{Concepts} \ \wedge \ c \in \textit{goalConcepts}(POs)\} \cup \\
&\quad \{pri(c,2) \mid c \in \textit{Concepts} \ \wedge \ c \in \textit{hypConcepts}(POs) \ \wedge \ c \notin \textit{goalConcepts(POs)}\} \cup \\
&\quad \{pri(c,3) \mid c \in \textit{Concepts} \ \wedge \ c \notin \textit{goalConcepts}(POs) \ \wedge \ c \notin \textit{hypConcepts}(POs)\} \cup \\
&\quad \{pri(c,4) \mid c \in \textit{goalNonCoreConcepts}(POs)\} \cup \\
&\quad \{pri(c,5) \mid c \in \textit{hypNonCoreConcepts}(POs) \ \wedge \ c \notin \textit{goalNonCoreConcepts}(POs)\}
\end{aligned}
$$

where:

- *pri(C, N)*: N denotes the priority assigned to concept C.
- *goalConcepts(POs)*: denotes core concepts occurring within the goals associated with *POs*.
- *hypConcepts(POs)*: denotes core concepts occurring within the hypotheses associated with *POs*.
- *goalNonCoreConcepts(POs)*: denotes non-core concepts occurring within the goals associated with *POs*.
- *hypNonCoreConcepts(POs)*: denotes non-core concepts occurring within the hypotheses associated with *POs*.

We have observed that in most cases we are able to identify the missing invariants by focusing in the first instance on the core concepts that arise within the goals of the failed POs; therefore the highest priority is given to such concepts. The core concepts associated with the hypotheses follow in order of interest while the remaining core concepts are dealt with next. Finally, compound expressions that occur within the goal (and hypotheses) which represent potential non-core concepts are identified. This is achieved by finding expressions within the POs which can be replicated via the application of PRs. We define these as "anticipated" non-core concepts. Note that we assume that the input POs are well defined since they are generated from the Rodin toolset; therefore, no invalid expressions would arise when identifying non-core concepts from the predicates of a PO. The implementation details of how these concepts are extracted from the POs is provided in Chapter 4.

Recall that the state in the example of Section 3.1.1 consisted of the following concepts:

full(A,B), x(A,B) and m(A,B)

where A represents the step in the simulation trace and B the value of the variable in the step. Moreover, the following failed PO is produced if the invariant is missing from the model:

$$x < m \vdash full = false.$$

Observe that a core concept can be identified from the goal of the PO, namely concept *full*, while there are two core concepts in the hypothesis, that is concepts x and m. Moreover, a non-core concept is found in the goal of the failed PO: *full=false*, which can be obtained through the application of the *split* PR over concept *full* with parameter *false*; i.e. *split⟨full,2=false⟩*. Moreover, a non-core concept is identified in the hypothesis of the PO, that is $x<m$. This concept can be replicated in HR through the application of the *numrelation* PR with parameter '<'; i.e. *numrelation⟨x,m,<⟩*. Therefore, the application of heuristic CH1 results in the following sets of prioritised core and non-core concepts:

$$\begin{aligned} \textit{prioritised core concepts} &= \{\textit{full}, x, m\} \\ \textit{prioritised non-core concepts} &= \{\textit{full=false}, x<m\} \end{aligned}$$

CH2. *Select the subset from the enabled PRs that are most relevant to the given failed POs.* Typically the invariants required in order to overcome proof failures have strong syntactic similarities with the failed POs. This is the intuition behind CH2, which selects PRs that focus HR's theory formation on such syntactic similarities. The selected PRs are identified as follows:

$$\begin{aligned} &\textit{select}(\textit{Enabled\_PRs}, \textit{POs}) \triangleq \\ &\quad \{pr \mid pr \in \textit{Enabled\_PRs} \ \wedge \ \exists\, op \in \textit{operators}(\textit{POs}) \,.\, op \in \textit{related\_ops}(pr)\} \end{aligned}$$

where:

- *operators(POs)*: denotes the set of operators that occur within the predicates associated with POs.
- *related_ops(PR)*: denotes the set of operators that are associated with PR (see Table 3.1 in Section 3.2.4).

Because of the set theoretic nature of Event-B, the compose, disjunct and negate production rules are always used in the search for invariants – where compose relates to conjunction and intersection, disjunct relates to disjunction and union and negate relates to negation and set complement.

Based on the failed PO shown in the previous heuristic, the application of CH2 to the

example produces the following selection of PRs:

$$\textit{production rules} = \{\textit{compose, disjunct, negate, split, numrelation}\}$$

the *compose*, *disjunct* and *negate* PRs are enabled by default, the *split* PR is selected because of the presence of the split value *false* in the goal of the PO, and the *numrelation* PR is selected because of the occurrence of operator < in the hypothesis.

As will be shown in Chapter 4, the empirical evidence we have gathered so far supports our rationale for both configuration heuristics.

3.3.2 Filtering heuristics

As mentioned earlier, HR produces a large number of conjectures, only a few of which represent candidate invariants. Constrained by the initial concept prioritisation, our FH heuristics guide the search for these candidate invariants via a series of 5 filters – defined here as a functional composition:

$$\textit{filter}(\textit{Model}, \textit{Conjectures}, \textit{POs}, \textit{PrioritisedConcepts}) \triangleq$$
$$\{c \mid c \in \textit{filter}_5(\textit{Model}, \textit{POs}, \textit{filter}_4(\textit{POs}, \textit{filter}_3(\textit{filter}_2(\textit{filter}_1(\textit{Conjectures}, \textit{PrioritisedConcepts})))))\}$$

Parameter *Model* refers to the Event-B model being analysed, *Conjectures* are all the conjectures generated through the theory formation process, *POs* is the set of failed proof obligations associated with the model and *PrioritisedConcepts* refers to the concepts identified through the application of configuration heuristic CH1.

As with the configuration heuristics, the development of the filtering heuristics was mainly driven by the PO failures; i.e. their focus is to find conjectures that would address the failures expressed by the POs. Furthermore, the order in which the heuristics are applied serve the purpose of optimising the search. That is, heuristics FH1 and FH2 require only a syntactic analysis of the conjectures, while heuristics FH3, FH4 and FH5 perform a semantic analysis through the use of external tools: the CVC3 prover and the Rodin toolset. This implies a major use of computational resources. Moreover, applying FH1 in the first place focuses the search on a smallest set of conjectures than applying FH2 in the first step. Additionally, the application of FH4 requires the translation of the conjectures from HR to Event-B; while FH1, FH2 and FH3 can be performed before this translation takes place. Finally, FH5 depends on the outcome of FH4; therefore, it must be applied in the last step. Note that applying these heuristics in a different order would yield the same result; however, the search would not be optimal.

Each of the 5 filtering heuristics is defined below:

FH1. *Select conjectures that focus on prioritised core and non-core concepts.*

As mentioned before, the required invariants tend to have strong syntactic similarities with the failed POs; therefore, focusing on the concepts associated with the POs increases the possibilities of discovering the missing invariants. In other words, focusing on the prioritised concepts identified through heuristic CH1; i.e. *PrioritisedConcepts*, guides the filtering process. This is achieved as follows:

$$
\begin{array}{l}
\mathit{filter}_1(\mathit{Conjectures}, \mathit{PrioritisedConcepts}) \triangleq \\
\{c \mid c \in \mathit{Conjectures} \ \wedge \ \exists\ \alpha \in \mathit{PrioritisedConcepts}\ . \\
\quad (c = (\alpha \Rightarrow \_) \ \vee \ c = (\_ \Rightarrow \alpha) \ \vee \ c = (\alpha \Leftrightarrow \_) \ \vee \ c = (\_ \Leftrightarrow \alpha) \ \vee \ c = \neg\, \exists(\_, \alpha, \_))\}
\end{array}
$$

The selected conjectures should focus purely on the prioritised concepts, i.e. we are interested only in equivalence and implication conjectures where either the left or right hand side represents a prioritised concept, as well as in all non-existence conjectures where a prioritised concept occurs.

Recall in the example the following prioritised core and non-core concepts were selected:

full, x, m, full=false and x<m

Applying heuristic FH1 requires the selection of conjectures that focus on these prioritised concepts. This results in a total of 14 conjectures associated with each concept as follows:

concept	equivalences	implications	non-exists
full	0	0	1
x	0	1	3
m	0	1	3
full=false	2	2	1
x<m	0	0	0

FH2. *Select conjectures where the sets of variables occurring on the left- and right-hand sides are disjoint.*
Based on our experience with Event-B, the literature and the case studies presented in this thesis, it was observed that typically the set of variables involved in an invariant does not contain duplicates. This heuristic focuses the search on conjectures that have such shape. Thus, the set of conjectures is further pruned by selecting only those that

do not contain multiple occurrences of a variable, i.e.

$$\begin{aligned}&\mathit{filter}_2(\mathit{Conjectures}) \triangleq \\ &\quad \{c \mid c \in \mathit{Conjectures} \ \wedge \ (((c = (L \Rightarrow R)) \ \vee \ c = (L \Leftrightarrow R)) \wedge \\ &\qquad \mathit{no\_duplicate\_variables}(L) \ \wedge \ \mathit{no\_duplicate\_variables}(R) \ \wedge \ \mathit{vars}(L) \cap \mathit{vars}(R) = \varnothing) \vee \\ &\qquad\quad (c = \neg\, \exists(\_) \ \wedge \ \mathit{no\_duplicate\_variables}(c)))\}\end{aligned}$$

where *vars(X)* denotes the free variables that occur in X, while the predicate *no_duplicate_variables(X)* holds if and only if no multiple occurrences of a variable occur within X. The disjointness property reflects the nature of gluing invariants which relate abstract and concrete variables.

In the case of the example, the application of the heuristic does not reduce the number of conjectures. This may occur because the model is very simple with only four steps in the simulation trace. Moreover, based on the new configuration applied through the CH heuristics only 23 conjectures are generated. This is contrary to the initial 2083 conjectures that are produced without tailoring the search in HR as was mentioned in Section 3.1.1.

FH3. *Select only the most general conjectures.*
This heuristic eliminates redundancies amongst the set of selected conjectures by removing those that are logically implied by more general conjectures, i.e.

$$\begin{aligned}&\mathit{filter}_3(\mathit{Conjectures}) \triangleq \\ &\quad \{c \mid c \in \mathit{Conjectures} \ \wedge \ \neg\, \exists\, c' \in (\mathit{Conjectures} \setminus \{c\}) \,.\, c' \Rightarrow c\}\end{aligned}$$

The identification of the most general conjectures is done through the use of CVC3; i.e. we use CVC3 to identify if $c' \Rightarrow c$.

Applying this heuristic to the example reduces the number of conjectures to half; i.e. a total of 7 conjectures, which are associated with the prioritised concepts as follows:

concept	equivalences	implications	non-exists
full	0	0	1
x	0	1	2
m	0	1	0
full=false	1	1	0
x<m	0	0	0

FH4. *Select conjectures that discharge the failed POs.*
Through this heuristic only conjectures that help discharge the given failed POs are

selected, i.e.

$$\begin{aligned} &\mathit{filter}_4(POs, Conjectures) \triangleq \\ &\quad \{c \mid c \in Conjectures \ \wedge \ \exists\, po \in POs \,.\, provable(c, po)\} \end{aligned}$$

where *provable(c, po)* holds if and only if po can be discharged by adding conjecture c to its set of hypotheses.

Applying this heuristic to the example results in the selection of only two conjectures associated with the prioritised non-core concept *full=false*, one equivalence and one implication.

FH5. *Select conjectures that minimise the number of additional proof failures that are introduced.*
Overcoming a proof failure potentially leads to new proof failures. Here we select conjectures that minimise the number of new failures introduced into the model, i.e.

$$\begin{aligned} &\mathit{filter}_5(Model, POs, Conjectures) \triangleq \\ &\quad \{c \mid c \in \bigcup_{po \in POs} minExtraFailedPOs(Model, po, Conjectures)\} \end{aligned}$$

where

$$\begin{aligned} &minExtraFailedPOs(M, P, C) \triangleq \\ &\quad \{c \mid c \in C \ \wedge \ provable(c, P) \wedge \\ &\qquad \forall\, c' \in (C \setminus \{c\}) \,.\, provable(c', P) \ \Rightarrow \mid failedPOs(c, M) \mid \leq \mid failedPOs(c', M)) \mid\} \end{aligned}$$

Note that *failedPOs(C, M)* denotes the set of failed POs that arise when conjecture C is added as an invariant to model M.

Regarding the example, the two conjectures selected by heuristic FH4 discharge the failed PO and do not produce any extra failure; thus, both of them are presented to the user as candidate invariants. The conjectures in their Event-B representation are:

$$\mathit{full}{=}\mathit{FALSE} \Leftrightarrow x{<}m \tag{3.9}$$

$$x \neq m \Rightarrow \mathit{full}{=}\mathit{FALSE} \tag{3.10}$$

Observe that conjecture (3.9) represents the invariant that was manually identified in Section 3.1.1. This is evidence of the effectiveness of the automatic approach presented here.

The application of the filtering heuristics described above may involve iteration. That is,

if focusing on the prioritised concepts identified from the goals of the failed POs does not generate suitable invariants, then the filtering heuristics are reapplied to the prioritised concepts that appear within the hypotheses. This iterative approach to discovering the missing invariants is typical of Event-B developments, as described in [22, Section 5] where invariant discovery is manual. However, when dealing with incorrect developments, this process will not terminate. Furthermore, the failure of a PO does not necessarily mean the absence of invariants in the model; a failure can also occur because the prover could not handle it. As a way of coping with these cases, the invariant discovery process would terminate if none of the selected conjectures help discharge at least one failed PO. This also means that a conjecture generated by HR, which is indeed a missing invariant, may be rejected through the approach because the auto-provers failed to discharge the associated POs. Currently, we do not have a mechanism to deal with this case. However, the use of a proof planner may help overcome this issue. This is part of our plan for future work. We return to this in Section 8.3.

3.4 Mondex invariant discovery revisited

In this section we show the application of the heuristics over the example of the Mondex system and the discovery of the missing gluing invariant:

$$\text{idleFP} = \text{statusF}^{-1}[\{\text{IDLEF}\}]$$

Without this invariant, an unprovable guard strengthening (*GRD*) PO[1], associated with event *StartFrom*, is generated. The unprovable PO, together with the abstract and concrete versions of the event *StartFrom*, are shown in Figure 3.20. Inspection of the PO shows that the abstract guard $p1 \in idleFP$ (goal) is not implied by the concrete guards (hypotheses).

Abstract Event	**Concrete Event**	**Failed PO**
StartFrom $\widehat{=}$ **any** t, p1 **where** p1 ∈ idleFP t ∈ startFromM p1 = from(t) Fseqno(t) = currentSeqNo(p1) **then** eprP := eprP ∪ {$p1$} idleFP := idleFP \ {$p1$} currentF2(p1) := t **end**	StartFrom $\widehat{=}$ **refines** StartFrom **any** t, p1 **where** p1 ↦ IDLEF ∈ statusF t ∈ startFromM p1 = from(t) Fseqno(t) = currentSeqNo(p1) **then** statusF(p1) := EPR currentF2(p1) := t **end**	p1 ↦ IDLEF ∈ statusF t ∈ startFromM p1 = from(t) Fseqno(t) = currentSeqNo(p1) ⊢ p1 ∈ idleFP

Figure 3.20: Failed guard strengthening PO resulting from a missing gluing invariant.

[1] A GRD PO verifies that the guards of a refined event imply the guards of the abstract event.

The application of the configuration heuristics starts with CH1, i.e. the identification of core and non-core concepts that occur within the failed PO in order to give them higher priority during theory formation. The following core concepts are identified from the goal and hypotheses of the failed PO:

{*idleFP, statusF, startFromM, from, FSeqno, currentSeqNo*}

Observe that the identifiers *p1* and *t*, which appear within the failed PO, are not considered as core concepts as they do not represent either a variable, a constant or a set, they represent parameters of the event. Furthermore, we identify the non-core concept:

$$\{p \mid p \in purses \wedge statusF(p) = IDLEF\}$$

because of the occurrence of constant *IDLEF*, which is a split value, and *statusF*, which is a core concept in the hypothesis:

$$p1 \mapsto IDLEF \in statusF$$

The non-core concept corresponds to all purses that are mapped by *statusF* onto *IDLEF*. This non-core concept can be obtained through the application of the split PR to the concept *statusF*, using the constant *IDLEF* as a parameter. No other non-core concepts are identified within the PO.

The next step is the application of heuristic CH2. The following PRs are selected for the invariant discovery process:

{*compose, disjunct, negate, split*}.

The compose, disjunct and negate PRs are always used in the search, as defined by heuristic CH2. The split PR is selected when any reference to a member of a finite set occurs. In the context of the example, the constant *IDLEF*, which is a member of the finite set *status*, triggers the selection of the split PR. Thus, the split PR is applied over the finite set *status* and the values to split are all the members of the set, i.e.: *IDLEF, EPR, EPA, ABORTEPR, ABORTEPA, ENDF, IDLET, EPV, ABORTEPV* and *ENDT*.

After completing the configuration, we run HR for 1000 steps which generates 2134 conjectures. This should be compared with the 4545 conjectures that are generated if our CH heuristics are not used to configure HR.

Next we follow with the application of the filtering heuristics focusing on the conjectures that relate to the prioritised core and non-core concepts. We focus first on the core and non-core concepts that appear within the goal of the failed PO. In our example, this implies looking for conjectures that involve the concept *idleFP*. The application of FH1 returns:

4 equivalences, 2 implications and 79 non-exists conjectures.

FH2 removes conjectures whose left- and right-hand sides are not disjoint with respect to the variable occurrences. This selection of conjectures produces:

1 equivalence, 2 implications and 79 non-exists conjectures

Less general conjectures are removed through FH3, this results in:

1 equivalence, 2 implications and 46 non-exists conjectures.

FH4 selects only conjectures that discharge the failed PO, resulting in:

1 equivalence, 0 implications and 0 non-exists conjectures

Note that in this example a single conjecture remains. Furthermore, this conjecture does not introduce any additional failures. The remaining conjecture is:

$$\forall A, B.(\textit{state}(A) \wedge \textit{purses}(B) \wedge \textit{idleFP}(A, B) \Leftrightarrow \textit{status}(\textit{IDLEF}) \wedge \textit{statusF}(A, B, \textit{IDLEF}))$$

which can be translated into the missing gluing invariant (3.4), i.e.:

$$idleFP = statusF^{-1}[\{IDLEF\}].$$

It should be noted that this conjecture was formed by HR after a single iteration of the theory formation process. This shows that in this example our heuristics guided HR to discover interesting conjectures early within the theory formation process.

The translation from HR generated conjectures to Event-B invariants is currently performed manually. In order to automate this process the first-order logic formulas used by HR must be translated into their equivalent set-theory representation in Event-B. In order to do this we need to remove extra type information and new concepts introduced by HR and HREMO, remove quantifiers whenever possible and identify the appropriate Event-B operators. For instance, a disjunction between two HR concepts that represent sets in the Event-B model is translated into set union. This translation is part of our future work.

3.5 Summary

This chapter presents an automatic approach to invariant discovery that builds upon HR, animation and proof-failure analysis. In particular, a set of heuristics are used to guide the search for invariants in HR. These heuristics exploit the strong interplay between modelling and reasoning in Event-B by using the feedback provided by failed POs to make decisions about how to configure HR. Specifically, the approach consists of analysing the structure of

failed POs to automate the prioritisation of concepts, the selection of PRs, and the filtering of conjectures. Two classes of heuristics are used to constrain the search for invariants: those used in configuring HR, i.e. configuration heuristics, and those used in filtering conjectures from HR's output, i.e. filtering heuristics. Using proof-failure analysis to prune the wealth of conjectures HR discovers, these heuristics have proven highly effective at identifying missing invariants.

Chapter 4

HRemo : Invariant discovery workbench and results

In this chapter we outline our prototype system, HREMO, which implements the approach described in the previous chapter. The HREMO system extends HR by implementing the configuration and filtering heuristics used for the discovery of invariants. The name HREMO reflects the extension to HR as well as the fact that it applies our reasoned modelling vision in which modelling and proof patterns are combined in order to provide high level modelling guidance to the user; in the case of HREMO this guidance takes the form of candidate invariants.

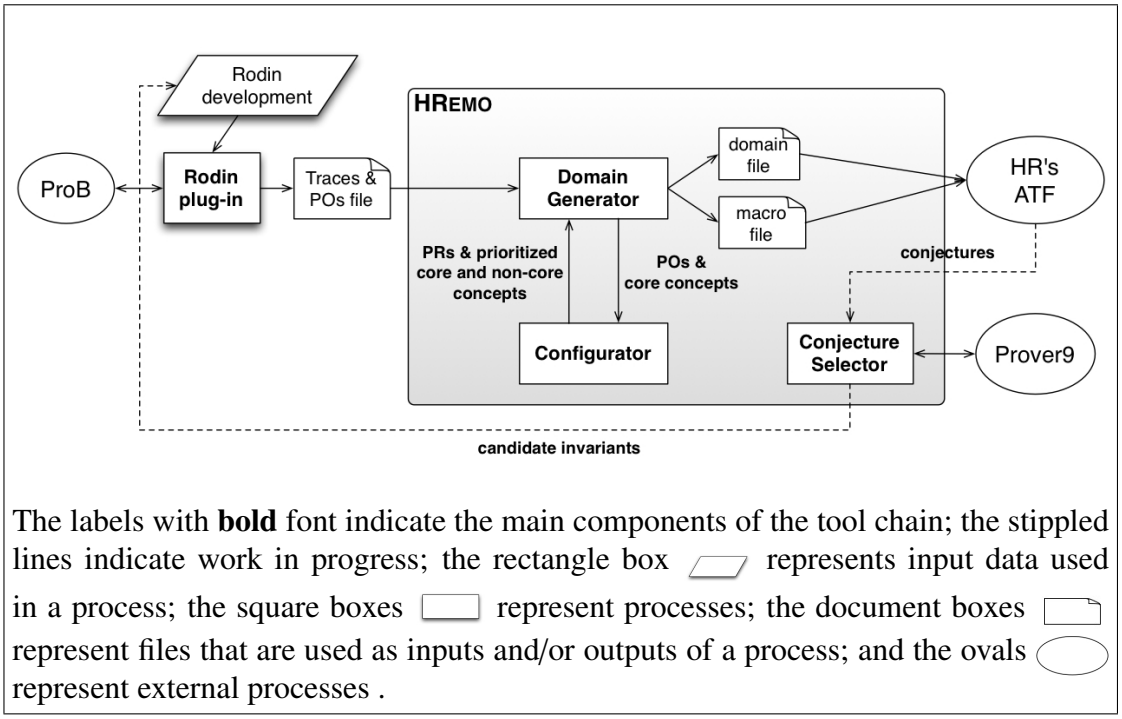
4.1 Tool architecture

The tool architecture is shown in Figure 4.1. While we have focused on Event-B, and the Rodin tool-set, our design aims to minimise the coupling between the formal modelling tool and HR. This was achieved by building a Rodin plug-in that manages the interface between an Event-B development and HR domain description. Our heuristics were mechanised via an extension to HR; i.e. HREMO. Both the HR tool and the Rodin toolset are implemented in Java; therefore, the implementation effort was also carried out in Java. Below we describe each of the major components of the tool.

4.1.1 Rodin plug-in

The main role of the *Rodin plug-in* is to provide an interface between the Rodin toolset and HREMO, specifically the plug-in is responsible for supplying domain information from an Event-B model, i.e. the animation traces and failed POs, to HREMO.

As shown in Figure 4.1, the plug-in receives as input a Rodin development. The user



The labels with **bold** font indicate the main components of the tool chain; the stippled lines indicate work in progress; the rectangle box represents input data used in a process; the square boxes represent processes; the document boxes represent files that are used as inputs and/or outputs of a process; and the ovals represent external processes .

Figure 4.1: Tool-chain architecture

selects an Event-B machine in the associated refinement chain for which the invariant discovery analysis should be performed – note that only one step of the refinement is handled at a time. ProB is then invoked in order to run a simulation of the selected refinement step. As previously mentioned, we rely on the random animation capability of ProB for the generation of traces; however, the plug-in allows the user to set up the initial state of the system and to perform his own animation steps in addition to the randomly generated steps. Currently, the plug-in sets up ProB to generate 100 random steps – In private discussions with Simon Colton, the creator of HR, 100 steps were identified as a sufficiently large data sample. Selecting fewer steps may not provide a representative sample of the behaviour of the system and selecting a larger number may affect the performance of HR; for instance, a PR may not be applicable when the number of tuples in the output data tables exceeds the maximum allowed by the PR – that is, there is a maximum size allowed for a concept data table in HR as a way of controlling possible memory issues.

When the simulation finishes, the plug-in extracts the structure of the model, i.e. variables, constants and sets, the examples from the animation traces and the failed POs. This information is then output into a xml file: the *traces and POs file*, which follows the Document Type Definition (DTD) schema shown in Appendix E.1 – an example of this output file is presented in Appendix E.2 for a simple traffic light model. The xml file produced by the plug-in contains:

- the elements of the model (i.e. variables, constants and sets) with their respective parameters;
- the value of each variable, constant and set for each step of the animation; and
- the failed POs represented as tree-like structures.

The tree-like representation of the POs classifies goal and hypothesis formula as either *binary*, *unary* or *literal*. Binary formulas are composed of a left- and a right-hand side formula and a binary operator. Unary formulas are composed by an unary operator and a formula, and literal formulas are composed of a string that represents a name or identifier. Examples of binary operators are $>, <, =, \in$ and $\subseteq$, while examples of unary operators are $\exists, \forall$ and $\neg$.

4.1.2 Domain Generator

The main functionality of the *domain generator* is to process the traces and POs in order to transform the information of the model into HR data tables. The input to this component is the *traces and POs file* and its outputs are a *domain file* that represents the model as domain information for HR, and a *macro file* which contains instructions about the application of the PRs as well as other parameters for the theory construction.

The traces contained in the file are transformed into background concepts in HR; i.e. the definition and data tables of each variable, constant and set are built as explained in Section 3.2.3. Parsing the POs consists of translating the tree-like representation explained in the previous section into a similar tree-like representation within HREMO; i.e. classifying each predicate in the PO as binary, unary and literal formulas. After the POs are parsed, the domain generator uses the *configurator* to analyse the failed POs. Examples of macro and domain files are given in Appendices E.4 and E.3, respectively.

4.1.3 Configurator

The *configurator* receives the parsed POs and the core background concepts from the domain generator. Its role is to apply the configuration heuristics in order to make the selection of the prioritised core and non-core concepts as well as the PRs from the failed POs. In other words, it determines the order of the concepts in the domain file, the non-core concepts that are forced into the domain, and the PRs that are to be specified in the macro file. The pseudo-code for the identification of core and non-core concepts is given in Algorithm 1. This procedure is broken into smaller pieces and explained below.

Algorithm 1 Pseudo-code for the identification of core and non-core concepts.

```
1  function coreAndNonCoreConcepts(formula)
2      if formula is a literal formula then
3          literal ← get identifier from formula
4          if literal is a domain variable OR literal is a domain constant then
5              add formula to prioritised core concepts
6              return TRUE
7      else if formula is a unary formula then
8          operator ← get operator from formula
9          uniFormula ← get unary sub-formula from formula
10         valid ← COREANDNONCORECONCEPTS(uniFormula)
11         if valid AND operator is compatible with a PR then
12             add formula to prioritised non-core concepts
13             return TRUE
14     else if formula is a binary formula then
15         operator ← get operator from formula
16         leftFormula ← get left sub-formula from formula
17         rightFormula ← get right sub-formula from formula
18         validLeft ← COREANDNONCORECONCEPTS(leftFormula)
19         validRight ← COREANDNONCORECONCEPTS(rightFormula)
20         if validLeft AND validRight AND operator is compatible with a PR then
21             add formula to prioritised non-core concepts
22             return TRUE
23         else
24             splitValues ← GETSPLITVALUES(formula)          ▷ see Algorithm 2
25             coreConcepts ← get core concepts from formula
26             if size of splitValues > 0 then
27                 for all val in splitValues do
28                     for all concept in coreConcepts do
29                         parameters ← get parameters from concept
30                         if val is compatible with parameters then
31                             binaryFormula ← newBinaryFormula(val, concept)
32                             add binaryFormula to prioritised non-core concepts
33     return FALSE
```

The analysis of this process is as follows:

1. The configurator analyses the goals and then the hypotheses within the failed POs. This is because as mentioned previously, we have observed that in most cases, we are able to identify the missing invariants by focusing in the first instance on the concepts that arise within the goals of the failed POs.

2. The configurator examines each formula to decide whether or not there exists a core or a non-core concept. This decision is made based on the following parameters:

   (a) A core concept is identified if a literal formula has been found (Line 2), and the identifier associated with the formula represents a variable or constant from the

domain background concepts (Line 4).

```
2  if formula is a literal formula then
3      literal ← get identifier from formula
4      if literal is a domain variable OR literal is a domain constant then
5          add formula to prioritised core concepts
6          return TRUE
```

(b) A non-core concept is a combination of variables and constants that can be replicated in HR through the use of the PRs. In order to identify non-core concepts from a formula we analyse the compatibilities between formula operators and PRs. These compatibilities were introduced in Table 3.1 (Section 3.2.4). A binary or unary formula is a valid non-core concept if its operator is compatible with a PR and its sub-formulas are themselves valid formulas. Therefore, for unary formulas (Line 7), the sub-formula is recursively analysed (Line 10) and if the sub-formula is valid and the operator is compatible with a PR (Line 11) the formula is added as a non-core concept.

```
7  if formula is a unary formula then
8      operator ← get operator from formula
9      uniFormula ← get unary sub-formula from formula
10     valid ← COREANDNONCORECONCEPTS(uniFormula)
11     if valid AND operator is compatible with a PR then
12         add formula to prioritised non-core concepts
13         return TRUE
```

Likewise, for binary formulas (Line 14), the left and right sub-formulas are recursively analysed (Lines 18 and 19) and if both sub-formulas are valid and the operator is compatible with a PR (Line 20) the formula is added as a non-core concept.

```
14 if formula is a binary formula then
15     operator ← get operator from formula
16     leftFormula ← get left sub-formula from formula
17     rightFormula ← get right sub-formula from formula
18     validLeft ← COREANDNONCORECONCEPTS(leftFormula)
19     validRight ← COREANDNONCORECONCEPTS(rightFormula)
20     if validLeft AND validRight AND operator is compatible with a PR then
21         add formula to prioritised non-core concepts
22         return TRUE
```

(c) Moreover, a non-core concept can be derived from the application of the split PR over a binary formula if:

i. the formula is not valid, i.e. one or both sub-formulas are not valid or the binary operator is not compatible with a PR, i.e. condition in Line 20 does not hold;

ii. the formula contains valid split values (Line 26) (procedure explained below in Algorithm 2); and

iii. the formula contains a core concept that is compatible with a split value; i.e. if one of the parameters of the core concept can be assigned the split value (Line 30); e.g. the core concept *ml_tl*, which denotes a traffic light, can be assigned the split value *green*.

```
24 splitValues ← GETSPLITVALUES(formula)                    ▷ see Algorithm 2
25 coreConcepts ← get core concepts from formula
26 if size of splitValues > 0 then
27     for all val in splitValues do
28         for all concept in coreConcepts do
29             parameters ← get parameters from concept
30             if val is compatible with parameters then
31                 binaryFormula ← newBinaryFormula(val, concept)
32                 add binaryFormula to prioritised non-core concepts
```

Algorithm 2 shows the pseudocode of the procedure that identifies split values from a formula. A valid split value is an identifier from a literal formula (Line 4) that does not represent a background concept (Line 5) but that is a member of a domain set (Line 7); e.g. the value *green* is a member of the domain set *Color*; therefore, *green* is a valid split value.

Algorithm 2 Pseudo-code for the identification of split values from a formula.

```
1  function GETSPLITVALUES(formula)
2      splitValues ← []                    ▷ stores the split values found in formula
3      if formula is a literal formula then
4          literal ← get identifier from formula
5          if literal not a domain variable AND literal not a domain constant then
6              for all set in domain sets do
7                  if literal is an element of set then
8                      add literal to splitValues
9      else if formula is a unary formula then
10         get split values from the unary formula in the input formula ...
11     else if formula is a binary formula then
12         get split values from the left and right formulas in the input formula ...
13     return splitValues
```

The same methodology is followed in order to identify the PRs that are used during the

search for conjectures within the theory formation process. That is, when an operator is found that is compatible with a PR, then the PR is enabled for the search.

4.1.4 Conjecture selector

The *conjecture selector* receives the set of conjectures generated by HR after the automated theory formation process. The main functionality of the selector is to prune the generated conjectures by selecting only those that represent candidate invariants. There are two important aspects to the implementation of the filtering process:

1. The identification of the *predecessors* of a concept. The predecessors are the core concepts involved in the construction of a concept generated through the PRs; i.e. the transitive closure of the ancestors relation. Algorithm 3 shows the pseudocode for the identification of the predecessors of a concept. If the concept represents a core concept (Line 2) its ID is returned; otherwise, the predecessors of the concept are the predecessors of its ancestor concepts; i.e. the concepts that were immediately used together with a PR in order to generate the concept within the theory (Lines 6-8).

Algorithm 3 Identification of the predecessors of a concept.

```
1 function PREDECESSORS(concept)
2     if concept is a core concept then
3         return id of concept
4     else
5         preds ← []                          ▷ stores the predecessors of concept
6         ancestorConcepts ← ancestors of concept
7         for all c in ancestorConcepts do
8             preds ← preds + PREDECESSORS(c)
9         return preds
```

2. The identification of *alternative conjectures*. When HR finds two equivalent concepts only one of them is kept in the theory since both concepts will derive the same conjectures. This results in some interesting implication conjectures not being identified because they were “hidden" to us by the equivalences previously formed by HR. In particular, interesting implications were not identified when a prioritised concept appeared in both sides of the conjecture. In other words, the invariant discovery process searches for conjectures of the form: $\alpha \Rightarrow \_$ and $\_ \Rightarrow \alpha$, where α is a prioritised concept. Assume that after the theory formation process is performed a conjecture $\beta \Rightarrow \alpha$ is found. Moreover, associated with concept β there is a conjecture $\beta \Leftrightarrow \delta$. Thus, $\delta \Rightarrow \alpha$ may be an interesting conjecture as well; however, this conjecture is not formed by HR because the equivalence conjecture “hides” δ from the theory – since HR chooses only one of the equivalent concepts to keep in the theory. Therefore,

the invariant discovery search must find all alternative conjectures that may represent potential candidate invariants.

Algorithm 4 shows the process of identifying alternative implication conjectures in which a prioritised concept appeared in the antecedent and consequent (Line 1). First, we find all equivalent concepts of the concept that is implied by or that implies the prioritised concept (Line 4). Then, if the prioritised concept is not a predecessor of the equivalent concept (Line 6), i.e. the concepts are disjoint, an implication conjecture is added to the set of interesting conjectures (either in the form '*prioritisedConcept* ⇒ *equivalentConcept*' (Line 8) or '*equivalentConcept* ⇒ *prioritisedConcept*' (Line 10)).

Algorithm 4 Identification of alternative conjectures of a prioritised concept.

```
1  Requires: (impConcept ⇒ prioritisedConcept OR prioritisedConcept ⇒ impConcept) AND
   (prioritisedConcept in predecessors(impConcept))
2  function ALTERNATIVECONJECTURES(prioritisedConcept, impConcept)
3      conjectures ← []                          ▷ stores the identified alternative conjectures
4      eqvConcepts ← get equivalent concepts of impConcept
5      for all eqv in eqvConcepts do
6          if prioritisedConcept not in PREDECESSORS(eqv) then
7              if prioritisedConcept ⇒ impConcept then
8                  imp ← (prioritisedConcept ⇒ eqv)
9              else
10                 imp ← (eqv ⇒ prioritisedConcept)
11             add imp to conjectures
12     return conjectures
```

Taking these two aspects into account, the filtering heuristics are applied as follows:

1. The selection of conjectures associated with prioritised core and non-core concepts is applied first. That is:

   (a) Finding the equivalence conjectures as shown in Algorithm 5. For each equivalence in the theory (Line 4), if the left- or the right-hand side concept of the conjecture is equal to the prioritised concept (Line 7) the equivalence is added to the set of interesting conjectures (Line 8).

   (b) Finding the non-existence conjectures as shown in Algorithm 6. If a prioritised concept occurs as a predecessor of a non-existence conjecture (Line 6), then the non-existence conjecture is added to the set of interesting conjectures (Line 7).

   (c) Identifying the implication conjectures requires:

      i. Finding the concepts for which the prioritised concepts are *generalisations*. HR defines a concept *c1* as a generalisation of a concept *c2* if the definition

Algorithm 5 Equivalence conjectures associated with prioritised core and non-core concepts.

```
1 function GETEQUIVALENCES(prioritisedConcept)
2     conjectures ← []                    ▷ stores the equivalence conjectures of prioritisedConcept
3     equivalences ← get equivalence conjectures from theory
4     for all eqv in equivalences do
5         left ← get left concept from eqv
6         right ← get right concept from eqv
7         if prioritisedConcept = left OR prioritisedConcept = right then
8             add eqv to conjectures
9     return conjectures
```

Algorithm 6 Non-existence conjectures associated with prioritised core and non-core concepts.

```
1 function GETNONEXISTENCECONJECTURES(prioritisedConcept)
2     conjectures ← []                    ▷ stores the non-existence conjectures of prioritisedConcept
3     nonExists ← get non-existence conjectures from theory
4     for all nEx in nonExists do
5         nExConcept ← get concept from nEx
6         if prioritisedConcept is in PREDECESSORS(nExConcept) then
7             add nEx to conjectures
8     return conjectures
```

of *c1* is contained within the definition of *c2*. In other words, generalisations are identified by comparing the definitions of the concepts, while the implication conjectures are identified by comparing their data tables. Algorithm 7 shows the process of identifying the generalisations associated with prioritised core and non-core concepts.

If a prioritised concept is a generalisation of a concept c (Line 6) and the prioritised concept is a predecessor of c (Line 7), then alternative conjectures between the prioritised concept and c are found (Line 8). However, if the prioritised concept is not a predecessor of c, i.e. the condition in Line 7 fails, then an implication conjecture of the form $c \Rightarrow$ *prioritisedConcept* is created (Line 11) and added to the collection of interesting conjectures (Line 12).

ii. Finding the implication conjectures associated with the prioritised concepts. This procedure is shown in Algorithm 8. If the left-hand side concept of an implication is equal to the prioritised concept (Line 7) and if the prioritised concept is a predecessor of the right-hand side concept (Line 8), the alternative conjectures of the prioritised concept and the right-hand side concept are found (Line 9). If both sides of the implication are disjoint, i.e. condition

Algorithm 7 Generalisations associated with prioritised core and non-core concepts.

```
1  function GETGENERALISATIONS(prioritisedConcept)
2      conjectures ← []                          ▷ stores the generalisations of prioritisedConcept
3      concepts ← get all concepts from theory
4      for all c in concepts do                                  ▷ Get the generalisations.
5          generalisations ← get generalisations from c
6          if prioritisedConcept in generalisations then
7              if prioritisedConcept is in PREDECESSORS(c) then
8                  altConjs ← ALTERNATIVECONJECTURES(prioritisedConcept, c)
9                  conjectures ← conjectures + altConjs
10             else
11                 imp ← c ⇒ prioritisedConcept
12                 add imp to conjectures
13     return conjectures
```

in Line 8 fails, the implication is added to the set of interesting conjectures (Line 12). A similar analysis is carried out if the prioritised concept is equal to the right-hand side of the implication (Line 13)

Algorithm 8 Implication conjectures associated with prioritised core and non-core concepts.

```
1  function GETIMPLICATIONS(prioritisedConcept)
2      conjectures ← []                          ▷ stores the implications of prioritisedConcept
3      implications ← get implication conjectures from theory          ▷ Get the implications
4      for all imp in implications do
5          left ← get left concept from imp
6          right ← get right concept from imp
7          if prioritisedConcept = left then
8              if prioritisedConcept is in PREDECESSORS(right) then
9                  altConjs ← ALTERNATIVECONJECTURES(prioritisedConcept, right)
10                 conjectures ← conjectures + altConjs
11             else
12                 add imp to conjectures
13         else if prioritisedConcept = right then
14             ...                                  ▷ similar analysis than previous case.
15     return conjectures
```

2. Next, the disjoint conjectures are selected. Algorithm 9 shows the pseudocode to define if a conjecture is disjoint. If the conjecture is an implication or an equivalence (Line 2) and, if there exists a core concept in the predecessors of the left-hand side concept that also occurs in the predecessors of the right-hand side concept and that represents a variable of the domain (Line 8) then the conjecture is not disjoint (Line 9). On the other hand, a non-existence conjecture (Line 10) is not disjoint if there exists a core concept that is a variable and that occurs more than once in its predecessors (Lines 15 and 16). The given conjecture is disjoint otherwise (Line 18).

Algorithm 9 Selection of disjoint conjectures.

```
1  function getDisjointConjectures(conjecture)
2      if conjecture is an implication OR conjecture is an equivalence then
3          leftConcept ← get left concept from conjecture
4          rightConcept ← get right concept from conjecture
5          leftIds ← PREDECESSORS(leftConcept)
6          rightIds ← PREDECESSORS(rightConcept)
7          for all left in leftIds do
8              if left in rightIds AND left in domain variables then
9                  return FALSE
10     else if conjecture is a non-existence conjecture then
11         concept ← get concept in conjecture
12         ids ← PREDECESSORS(concept)
13         for all id1 in ids do
14             for all id2 in ids do
15                 if id1 = id2 AND id1 position is different to id2 position AND
16               id1 in domainVariables then
17                     return FALSE
18     return TRUE
```

3. Prover9 [94] is used to choose the most general conjectures. Prover9 is an automated theorem prover for first-order logic with equality and is the successor of Otter [93], the theorem prover originally used by HR to prove/disprove the conjectures. So, for two conjectures *conj1* and *conj2*, if *conj2* is logically implied by *conj1*, *conj2* can be removed from the set of candidate invariants.

4. Selecting the conjectures that discharge the failed POs (FH4) and that produce less extra failed POs (FH5) are the next steps of the process. We have implemented heuristics FH1, FH2 and FH3; however, although we believe the approach can be applied to different formalisms, the implementation of heuristics FH4 and FH5 are language- and tool-dependent; this is due to the fact that the generation of POs in a formal model requires specialised mechanisms that are particular to each formalism. Therefore, heuristics FH4 and FH5 are currently performed manually. Their implementation for Event-B would involve translating the conjectures selected up to heuristic FH3 from the format given by HR to the Event-B language. This translation will be pursued in future work.

The output from the conjecture selector is the set of candidate invariants.

4.2 Experimental results

The experiments we carried out were divided into two stages. The first stage involved the *development* of our heuristics, and was based upon four relatively simple Event-B models, as described below:

1. *Traffic light system:* this model represents a traffic light circuit that controls the sequencing of lights. It is composed of an abstract model and involves a single refinement. The abstract model controls the red and green lights, while the refinement introduces a third light to the sequence, i.e. an amber light.

2. Two representations of a vending machine:

   - *Set representation:* this model of a vending machine controls the stock of products through the use of states. It is composed of an abstract and a concrete model. The abstract model represents the states of products using state sets, while the refinement introduces a status function that maps products to their states.
   - *Arithmetic representation:* this model of the vending machine uses natural numbers to represent the stock and money held within the machine. While the abstract model deals with a single product, the refinement introduces a second product to the vending machine.

3. *Refinements one and two of Abrial's cars on a bridge system [3]:* models a system that controls the flow of cars on a bridge that connects a mainland to an island. At the abstract level, cars are modelled leaving and entering the island. The first refinement introduces the requirement that the bridge only supports one way traffic, while the second refinement introduces traffic lights.

All the configuration and filtering heuristics were identified in the first stage. We used the second stage of our experiments to *evaluate* these heuristics; i.e. during the second stage no new heuristics were developed, but the experiments were used to provide evidence of the effective application of the approach. Here the experiments were performed on more complex Event-B models:

1. *Refinement three of Abrial's cars on a bridge system [3]:* the third refinement of this system models the introduction of sensors that detect the physical presence of cars.

2. *The Mondex system [22]:* models an electronic purse that allows the transfer of money between purses. This development is composed of one abstract model and nine refinement steps. We applied our invariant discovery technique over each of the these refinement steps.

3. *Location access controller system [3]:* models a system that controls the access to the rooms within a building.

4. *Flash-based file system [39]:* models a flash-based file system that allows a user to read, write and erase information from a flash disk. This development consists of two models; one that handles the logical (software) operation and the other that handles the physical (hardware) operation. Here we targeted the latter, which involves five refinement steps that model the reading, writing and erasing of the physical pages within a flash disk.

In the following section we present the results obtained by applying our approach to the refinement of level three of the Mondex system. Furthermore, in Sections 4.2.1.1 and 4.2.2 we summarise the results we achieved across the whole refinement chain of the Mondex system model and the flash file system case study.

4.2.1 The Mondex system

In the work reported in [22] on the Mondex system, it was highlighted that the manual analysis of failed POs was used to guide the construction of gluing invariants. In particular, this was illustrated in the third step of the refinement in which, through the analysis of failed POs, and after three iterations of invariant strengthening, the set of invariants needed to prove the refinement between levels three and four were added to the model. As part of our experiments, we re-constructed the Mondex system in Event-B based upon the development presented in [22]. In this section we show that these results are similar to the ones obtained through the interactive development in [22].

In level three of the Mondex system a transaction is permitted to be in one of four states: *idle, pending, recover or ended*. The refinement in level four introduces dual states to a transaction so that each side has their own local protocol state. That is: states *idleF, epr, epa, abortepr, abortepa* and *endF* for the source side of a transaction, and states *idleT, epv, abortepv* and *endT* for the target side. These states model the communication between the source and the target sides during a transaction; e.g. expecting request (epr) or expecting value (epv), etc[1]. In order to evaluate our approach, we introduced the model in level four with only basic typing invariants. The absence of the rest of the invariants produces the failed POs shown in Figure 4.2. In order to discharge these failed POs, the gluing invariants that relate the abstract states with the states introduced in the refinement are needed.

HREMO start the invariant discovery process with the application of heuristic CH1. The set of core concepts selected from the failed POs are:

[1]Further details about the model of the Mondex system can be found in [22].

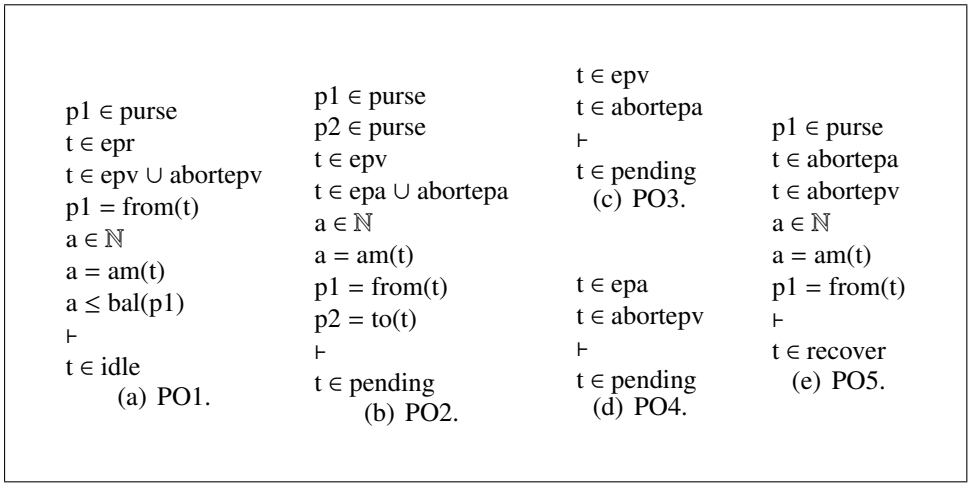


Figure 4.2: First set of failed POs.

{*idle, pending, recover, purse, epr, epv, abortepv, from, am, bal, epa, abortepa, to*}

Note that *t*, *a*, *p1* and *p2* do not represent core concepts since they are parameters of events in the model. Moreover, from the analysis of the predicates in the failed POs, HREMO identifies the following non-core concepts:

{*epv* ∪ *abortepv, epa* ∪ *abortepa*}

These concepts are identified from hypotheses $t \in epv \cup abortepv$ and $t \in epa \cup abortepa$ within PO1 and PO2, respectively. Figure 4.3 illustrates how $epv \cup abortepv$ is identified as a non-core concept from the formula $t \in epv \cup abortepv$ (based on Algorithm 1 described previously in Section 4.1.3). The formula tree is traversed until finding the leaves (Figure 4.3(a)), then each node is analysed returning true if the node is a valid formula, i.e. a core or a non-core concept, or a compatible operator. It returns false otherwise. Note that the left sub-formula is not a valid core or non-core concept (Figure 4.3(b)). This is because *t* does not represent a concept in the domain, i.e. it represents a parameter of the event associated with the failed PO. On the contrary, *epv* and *abortepv* are valid core concepts (Figure 4.3(c) and Figure 4.3(d)) as they are variables in the domain. Furthermore, ∪ is a valid operator (Figure 4.3(e)) since it is compatible with the *disjunct* PR as stated in Table 3.1. This results in the right sub-formula being identified as a valid non-core concept (Figure 4.3(f)). Finally, the operator ∈ is not valid (Figure 4.3(g)) as it is not compatible with any PR. For this reason, and because the left sub-formula is also invalid, the formula as a whole is not considered as a valid non-core concept (Figure 4.3(h)). A similar analysis is carried out over formula $t \in epa \cup abortepa$.

The process continues with the selection of the PRs. Based on the failed POs shown in Figure 4.2, the following PRs are selected for the search:

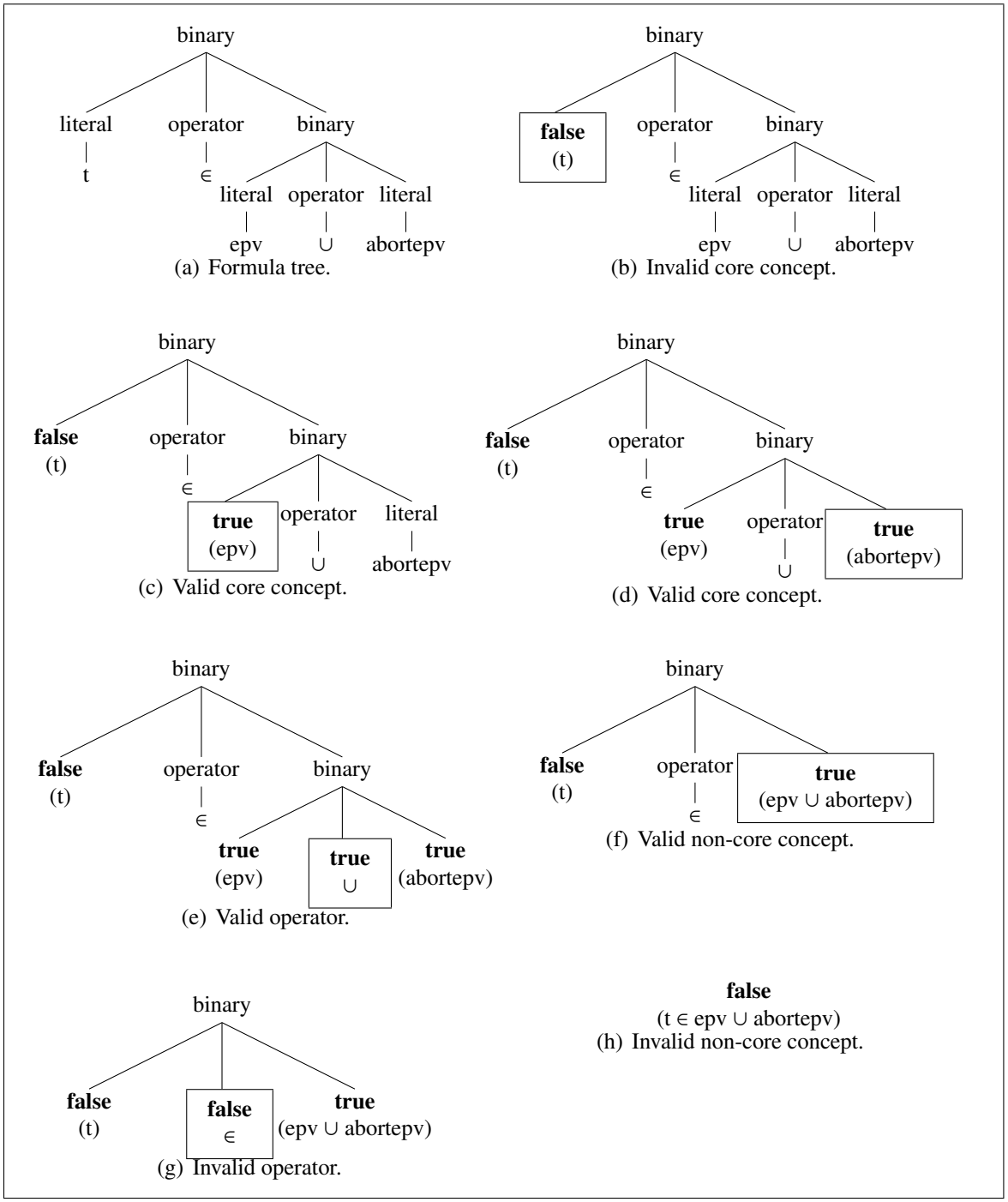


Figure 4.3: Identifying core and non-core concepts from formula t ∈ epv ∪ abortepv.

{*compose, disjunct, negate, numrelation*}

The compose, disjunct and negate production rules are always used in the search as stated in heuristic CH2. The numrelation PR is selected because of the occurrence of operator ≤ in hypothesis $a \leq bal(p1)$ within PO1. After the configuration heuristics have been applied HR is run for 1000 steps.

The filtering heuristics are now applied. Heuristic FH1 suggests looking at the prioritised concepts associated with the goals of the failed POs. From the goals of the POs shown in Figure 4.2, the concepts *idle, pending* and *recover* are identified. Thus, HREMO looks for the conjectures associated with each of these concepts. The results of applying FH1 are shown in Table 4.1, along with the results obtained for heuristics FH2, FH3 and FH4 for each of the selected concepts.

Concept	Heuristic	Equivalences	Implications	Non-exists	Total
idle	FH1	7	27	24	**58**
	FH2	0	27	24	**51**
	FH3	0	16	17	**33**
	FH4	0	3	0	**3**
pending	FH1	6	27	35	**68**
	FH2	0	27	35	**62**
	FH3	0	8	26	**34**
	FH4	0	2	0	**2**
recover	FH1	9	51	41	**101**
	FH2	2	51	41	**94**
	FH3	2	3	30	**35**
	FH4	1	0	0	**1**

Table 4.1: Results of the application of filtering heuristics FH1, FH2, FH3 and FH4.

As can be observed, after applying the four initial filtering heuristics we have narrowed the set of selected conjectures to six: three implications involving the concept *idle*, two implications for concept *pending* and one equivalence about the concept *recover*. These conjectures are presented in Table 4.2.

Conjecture	Discharged POs	Extra failed POs
1. abortepr $\cup$ epr $\subseteq$ idle	PO1	2
2. idleF $\cup$ epr $\subseteq$ idle	PO1	1
3. idleT $\cup$ epr $\subseteq$ idle	PO1	2
4. epv $\cap$ (epa $\cup$ abortepa) $\subseteq$ pending	PO2, PO3	2
5. epa $\cap$ (epv $\cup$ abortepv) $\subseteq$ pending	PO4	3
6. recover = abortepv $\cap$ abortepa	PO5	0

Table 4.2: Conjectures obtained after applying selection heuristic FH4.

Note that we have given the equivalent set theoretic representation of the conjectures instead of using the universally quantified format provided by HR. This is because some

experiments, for instance the development of the Mondex system carried out in [22], have shown that the automatic provers do better with quantifier-free predicates.

Heuristic FH5 is the final step in the discovery process. This heuristic selects the conjectures that produce the smaller number of new failed POs. The correspondence between the conjectures and the failures they help overcome is presented in Table 4.2 as well as the number of extra failed POs generated when they are introduced into the model.

As can be observed, the three conjectures associated with concept *idle*, i.e. conjectures 1,2 and 3, discharge PO1. However, two of them each generate two new failed POs, while the other conjecture only generates one extra failure. Thus, conjecture 2, which produces the least number of failures, is presented as a candidate invariant. Regarding the two conjectures associated with concept *pending*, one of them discharges PO2 and PO3 and produces two new failed POs, while the other one discharges PO4 but produces three new failed POs. As there are no other conjectures that help overcome the failures produced by PO2, PO3 and PO4, both conjectures are suggested as candidate invariants. Finally, the equivalence conjecture associated with concept *recover* discharges PO5 and it does not produce any extra failures, so this conjecture is also suggested as a candidate invariant. Thus, the candidate invariants obtained from the first iteration of HREMO over the third refinement of the Mondex system are shown in Figure 4.4.

$$
\begin{array}{l}
(\mathrm{idleF} \cup \mathrm{epr}) \subseteq \mathrm{idle} \\
\mathrm{epv} \cap (\mathrm{epa} \cup \mathrm{abortepa}) \subseteq \mathrm{pending} \\
\mathrm{epa} \cap (\mathrm{epv} \cup \mathrm{abortepv}) \subseteq \mathrm{pending} \\
\mathrm{recover} = \mathrm{abortepa} \cap \mathrm{abortepv}
\end{array}
$$

Figure 4.4: Candidate invariants obtained from first iteration.

An overview of the application of the heuristics is presented in Table 4.3. Note that this is a summary of the results shown in Table 4.1. Initially there are a total of 7296 conjectures generated by HR; after applying heuristic FH1 the set of conjectures is reduced to a total of 227 conjectures. This represents a 97% reduction of the total of conjectures generated by HR, proving heuristic FH1 to be effective at focusing the search for candidate invariants. Heuristic FH2 reduces the set of conjectures from 227 to 207; i.e. 8% of conjectures are discarded. This may suggest that heuristic FH2 could be removed from the process since the reduction is not significant; however, it is possible that for larger systems this heuristic proves more effective. Further experiments would be required in order to identify the effectiveness of the heuristic in larger case studies. This study is not carried out in this thesis but it is part of our future work. Heuristics FH3 and FH4 decrease the number of conjectures to 102 and 6, respectively. This represents a reduction of 51% by FH3 and 94% by FH4. Both heuristics decrease considerably the number of conjectures; in particular, heuristic FH4 shows that, assuming the model is correct, using the failed POs as a measure to discard conjectures is

very effective. Finally, heuristic FH5 results in a total of 4 conjectures. At this point the reduction is minimal since FH4 has focused the search to only those conjectures that help discharge the failed POs. Nevertheless, the application of heuristic FH5 is very important since decreasing the number of extra failures results in less interactions with HREMO.

	idle	**pending**	**recover**	**total**
FH1	58	68	101	**227**
FH2	51	62	94	**207**
FH3	33	34	35	**102**
FH4	3	2	1	**6**
FH5	1	2	1	**4**

Table 4.3: Overview of heuristics application.

After the new set of invariants is introduced into the model, six new PO failures are generated. The new set of failed POs is shown in Figure 4.5. These POs raise failures related to the preservation of the invariants found in the first iteration. A second iteration of the invariant discovery process is then performed based on the new set of failed POs. The application of the configuration heuristics results in the following prioritised core and non-core concepts:

$$\begin{aligned} \textit{core concepts} &= \{\textit{epv, epa, abortepa, pending, abortepv, epr,} \\ &\quad \textit{idleF, idle, purse, idleT, to, from, am, bal}\} \\ \textit{non-core concepts} &= \{\textit{epa} \cup \textit{abortepa, epv} \cap \textit{(epa} \cup \textit{abortepa),} \\ &\quad \textit{epv} \cup \textit{abortepv, epa} \cap \textit{(epv} \cup \textit{abortepv)}\} \end{aligned}$$

and in the selection of the following PRs:

$$\textit{production rules} = \{\textit{compose, disjunct, negate, numRelation}\}.$$

As with the first iteration, we apply the filtering heuristics first to the core concepts identified in the goals of the failed POs. Therefore, the search for invariants is focused on concepts *epv, epa, abortepa, pending, abortepv, idleF, epr* and *idle*. A summary of the result of the application of heuristics FH1 to FH4 is shown in Table 4.4. Observe that after applying the four initial filtering heuristics, the set of selected conjectures has been narrowed to a total of four non-existence conjectures: one involving the concept *epv*, two for concept *epa* and one about the concept *idleF*. These conjectures are presented in Table 4.5. Note that these conjectures help overcome all the current failures, i.e. PO6, PO7, PO8, PO9, PO10 and PO11; therefore, the four conjectures are proposed as candidate invariants. Note also that when the new set of invariants are added to the model, two new failed POs are generated. These new failures are shown in Figure 4.6.

$epv \cap (epa \cup abortepa) \subseteq pending$
$p2 \in purse$
$t \in idleT$
$p2 = to(t)$
$\vdash$
$(epv \cup \{t\}) \cap (epa \cup abortepa) \subseteq pending$

(a) PO6.

$epa \cap (epv \cup abortepv) \subseteq pending$
$p2 \in purse$
$t \in idleT$
$p2 = to(t)$
$\vdash$
$epa \cap ((epv \cup \{t\}) \cup abortepv) \subseteq pending$

(b) PO7.

$epr \cup idleF \subseteq idle$
$p1 \in purse$
$t \in epr$
$t \in epv \cup abortepv$
$p1 = from(t)$
$a \in \mathbb{N}$
$a = am(t)$
$a \leq bal(p1)$
$\vdash$
$(epr \setminus \{t\}) \cup idleF \subseteq idle \setminus \{t\}$

(c) PO8.

$epa \cap (epv \cup abortepv) \subseteq pending$
$p1 \in purse$
$p2 \in purse$
$t \in epv$
$t \in epa \cup abortepa$
$a \in \mathbb{N}$
$a = am(t)$
$p1 = from(t)$
$to(t) = p2$
$\vdash$
$epa \cap ((epv \setminus \{t\}) \cup abortepv) \subseteq pending \setminus \{t\}$

(d) PO9.

$epa \cap (epv \cup abortepv) \subseteq pending$
$t \in epv$
$t \in abortepa$
$\vdash$
$epa \cap ((epv \setminus \{t\}) \cup (abortepv \cup \{t\})) \subseteq pending \setminus \{t\}$

(e) PO10.

$epv \cap (epa \cup abortepa) \subseteq pending$
$t \in epa$
$t \in abortepv$
$\vdash$
$epv \cap ((epa \setminus \{t\}) \cup (abortepa \cup \{t\})) \subseteq pending \setminus \{t\}$

(f) PO11.

Figure 4.5: Second set of failed POs.

Concept	**Equivalences**	**Implications**	**Non-exists**
epv	0	0	1
epa	0	0	2
abortepa	0	0	0
pending	0	0	0
abortepv	0	0	0
idleF	0	0	1
epr	0	0	0
idle	0	0	0

Table 4.4: Iteration 2: Results of the application of filtering heuristics FH1-FH4.

Conjecture	Discharged POs	Extra failed POs
epv ∩ abortepv = ∅	PO9, PO11	1
epa ∩ abortepa = ∅	PO10	0
idleT ∩ (epa ∪ abortepa) = ∅	PO6, PO7	1
idleF ∩ epr = ∅	PO8	0

Table 4.5: Iteration 2: Conjectures obtained after applying heuristics FH1-FH4.

epv ∩ abortepv = ∅
p2 ∈ purse
t ∈ idleT
p2 = to(t)
⊢
(epv ∪ {t}) ∩ abortepv = ∅

(a) PO12.

idleT ∩ (epa ∪ abortepa) = ∅
p1 ∈ purse
t ∈ epr
t ∈ epv ∪ abortepv
p1 = from(t)
a ∈ ℕ
a = am(t)
a ≤ bal(p1)
⊢
idleT ∩ ((epa ∪ {t}) ∪ abortepa) = ∅

(b) PO13.

Figure 4.6: Third set of failed POs.

As there are still failures present in the model, another iteration of HREMO is required. Again, applying the configuration heuristics results in the selection of the following core and non-core concepts:

$$\begin{aligned} \textit{core concepts} &= \{\textit{epv, abortepv, idleT, epa, abortepa, purse,} \\ &\quad \textit{to, epr, from, am, bal}\} \\ \textit{non-core concepts} &= \{\textit{epv} \cap \textit{abortepv, epa} \cup \textit{abortepa,} \\ &\quad \textit{idleT} \cap \textit{(epa} \cup \textit{abortepa), epv} \cup \textit{abortepv}\} \end{aligned}$$

and in the selection of the following PRs:

$$\textit{production rules} = \{\textit{compose, disjunct, negate, numRelation}\}.$$

As with the first and second iteration, the filtering heuristics are applied in the first place to the core concepts identified in the goals of the failed POs. In this case, the search for invariants is focused on concepts *epv, abortepv, idleT, epa* and *abortepa*. The result of applying heuristics FH1-FH4 is shown in Table 4.6. Observe that the application of these filtering heuristics have limited the set of selected conjectures to a total of one non-existence conjecture involving the concept *epv*. This conjecture discharges both PO12 and PO13 and it does not generate any extra failed PO; thus, this conjecture is proposed as a candidate invariant.

Concept	Equivalences	Implications	Non-exists
epv	0	0	1
abortepv	0	0	0
idleT	0	0	0
epa	0	0	0
abortepa	0	0	0

Table 4.6: Iteration 3: Results of the application of filtering heuristics FH1-FH4.

As no extra failures are generated with the introduction of this invariant, no further iterations are needed. The final set of invariants represented by the conjectures obtained from HREMO in the three iterations of our approach are shown in Figure 4.7.

$(\text{idleF} \cup \text{epr}) \subseteq \text{idle}$
$\text{epv} \cap (\text{epa} \cup \text{abortepa}) \subseteq \text{pending}$
$\text{epa} \cap (\text{epv} \cup \text{abortepv}) \subseteq \text{pending}$
$\text{recover} = \text{abortepa} \cap \text{abortepv}$

(a) First iteration.

$\text{epv} \cap \text{abortepv} = \varnothing$
$\text{epa} \cap \text{abortepa} = \varnothing$
$\text{idleT} \cap (\text{epa} \cup \text{abortepa}) = \varnothing$
$\text{epr} \cap \text{idleF} = \varnothing$

(b) Second Iteration.

$\text{idleT} \cap (\text{epv} \cup \text{abortepv}) = \varnothing$

(c) Third iteration.

Figure 4.7: Mondex refinement fourth: automatically discovered invariants.

The invariants shown in Figure 4.7 for level 4 of the Mondex case study are a subset of the invariants suggested in [22] for this step of the refinement. In total HREMO obtained 8 invariants from the 17 used in [22], plus one invariant that implies two of the invariants suggested in [22]; that is, HREMO identified invariant *(idleF ∪ epr) ⊆ idle* which implies invariants *idleF ⊆ idle* and *epr ⊆ idle* suggested in [22]. It is important to note that HREMO has addressed all the failures produced in the refinement step. Some of the extra invariants used in [22] represent new requirements of the system, which are beyond the scope of our technique since we only target invariants needed to prove the refinement steps. However, some of these invariants are required in order to prove later refinements. This means that some extra failures would arise in the subsequent refinements. HREMO can be applied to explore these failures.

4.2.1.1 Overview of the application of HREMO to the full refinement chain of the Mondex case study

In the previous section we detailed the application of our approach to level four of the Mondex development. In Table 4.7 we summarise the results of applying the invariant discovery

technique to each step of the refinement chain. As can be observed our technique succeeded in finding the invariants required to discharge all failed POs in levels four, five, six and nine. However, in levels two, three, seven and eight not all failures were discharged. Next, we explain for each level why the missing invariants were not identified by HREMO.

Step	Discovered invariants			All failures discharged?
	Glue	System	Total	
Level 2	0	1	**1**	No
Level 3	0	0	**0**	No
Level 4	4	5	**9**	Yes
Level 5	0	9	**9**	Yes
Level 6	7	45	**52**	Yes
Level 7	3	0	**3**	No
Level 8	0	0	**0**	No
Level 9	10	0	**10**	Yes

Table 4.7: Summary of results for the Mondex development.

The following invariants were not identified at level 2:

$$\forall\, p \cdot p \in purse \Rightarrow abal(p) = bal(p) + sum(am[pfrom[\{p\}]]) \tag{4.1}$$

$$\forall\, p \cdot p \in purse \Rightarrow lost(p) = sum(am[lfrom[\{p\}]]) \tag{4.2}$$

which are gluing invariants that explain how the balance and the money of failed transactions are related in the abstract and concrete level – where the abstract balance of a purse, i.e. *abal*, is equal to the concrete balance of the purse, i.e. *bal*, plus the amount of money involved in all pending transactions for which the purse is the source (a similar case is represented by the other invariant). The reason these invariants are not identified is because the constant *sum* cannot be represented as a concept within HR. Note that *sum* represents a partial function which maps finite sets of natural numbers to natural numbers that represent the summation of each set, i.e:

$$\text{sum} \in \mathbb{P}(\mathbb{N}) \nrightarrow \mathbb{N},$$

for instance, *sum({ 1, 2, 3 }) = 6*. This constant would be defined within HR as:

sum(A,B)

where *A* is of type $\mathbb{P}(\mathbb{N})$ and *B* is of type $\mathbb{N}$. This means that the first parameter of *sum* is a set; however, HR does not support sets as parameters of concepts, it only accepts parameters that contain at most one element.

The following invariants are the missing invariants from level 3:

$$pfrom^{-1} = (pending \lhd from) \tag{4.3}$$

$$lfrom^{-1} = (recover \lhd from) \tag{4.4}$$

These invariants specify how the redundant information provided by variables *pfrom* and *lfrom* can be obtained by restricting the domain of function *from* with the set of transactions in the *pending* and *recover* states, respectively. Note that the functions *pfrom*, *lfrom* and *from* are defined as follows:

$$\text{pfrom} \in \text{purse} \leftrightarrow \text{trans}$$
$$\text{lfrom} \in \text{purse} \leftrightarrow \text{trans}$$
$$\text{from} \in \text{trans} \rightarrow \text{purse}$$

The difficulty with these invariants is that the inverse type of a concept cannot be generated via HR because there are no PRs that allow the permutation of columns within a data table. As a consequence, although HR invents the concepts on the right-hand side of the invariants, i.e. *pending* $\lhd$ *from* and *recover* $\lhd$ *from*, it cannot invent the concepts on the left-hand side, i.e. $pfrom^{-1}$ and $lfrom^{-1}$. We therefore consider that a PR that permutes columns within a data table would be useful addition to HR for use in the formal modelling domains, such as Event-B.

Regarding level 7, the invariants that could not be discovered by HREMO are presented below:

$$\forall\, p \cdot p \in (active \setminus idleFP) \land p \in dom(currentF2) \Rightarrow currentF(p) = currentF2(p) \tag{4.5}$$

$$\forall\, p \cdot p \in (active \setminus idleTP) \land p \in dom(currentT2) \Rightarrow currentT(p) = currentT2(p) \tag{4.6}$$

$$\forall\, p, t \cdot p \in active \land from(t) = p \land currentSeqNo(p) = Fseqno(t) \Leftrightarrow currentF(p) = t \tag{4.7}$$

$$\forall\, p, t \cdot p \in active \land to(t) = p \land currentSeqNo(p) = Tseqno(t) \Leftrightarrow currentT(p) = t \tag{4.8}$$

All these are gluing invariants that describe how the abstract variables *currentF* and *currentT* are represented in the concrete level. Invariants (4.5) and (4.6) are not identified through HREMO because they conflict with heuristic FH2 since the left- and right-hand sides of the conjectures are not disjoint with respect to variables *currentF2* and *currentT2*. On the other hand, invariants (4.7) and (4.8) present the same problem that arose with invariants (4.3) and (4.4) at level 3. Which again suggests that a new PR, that permutes the columns within a data table, would be worth investigating in the future.

The final missing invariant is required in level 8, and takes the form:

$$\forall\, p, n \cdot (p \mapsto n \in used \Rightarrow n \leq currentSeqNo(p)) \tag{4.9}$$

which represents a gluing invariant that ensures the freshness of transactions is consistent at the abstract and concrete levels by using sequence numbers.

This invariant cannot be invented by HR, because concepts *used* and *currentSeqNo* do not meet the conditions for the application of the *numrelation* PR. In particular, the *numrelation* PR can only be applied to concepts of arity 2, whereas the relevant concepts in this example have arity 3. The rationale behind the decision to impose an arity limit was to avoid unnecessarily broadening the search space, since the *exists* PR can be applied prior to *numrelation*, in order to reduce a concept's arity. However, in this case, applying the *exists* PR would reduce in a loss of information which is necessary in order to form the invariant. We propose that this could be amended simply, by enabling the *numrelation* PR to operate on concepts of n arity where n is a user-defined parameter, with a default setting of 2, which can be set prior to a run.

4.2.2 Flash file case study

This case study [39] was motivated by a "verification grand challenge" posed by Holzmann [65]. As mentioned previously, this development consists of two sub-models; one that handles the software component of the flash file system and one that handles the hardware component. In this section we show the application of HREMO to the hardware model. The refinement chain for this model consists of five steps:

Initial model: introduces an abstract representation of the flash device as an array of pages that store data. It also adds abstract events for reading, writing and erasing pages.

Level two: introduces the concept of a page register that is used as an intermediate storage when reading and writing data from/to a physical page of the flash disk.

Level three: models the process of block reclamation which consists of erasing and reusing a set of pages from the flash disk.

Level four: handles the relocation of pages from a block that is going to be used to a free block.

Level five: refines the process of erasing a block from the flash disk.

Level six: introduces the concept of a status register which is used to indicate if the flash device is ready and if the previous operation has succeeded or not.

Table 4.8 summarises the results of the invariant discovery process carried out by HREMO. Note that at levels two and four, HREMO found a set of invariants that discharged all the failed POs. At level three, however, not all the failures were discharged. Note also that when

including only typing invariants at levels five and six, the models did not generate any failed POs. This meant that HREMO was not required or Not Applicable (NA) for the refinement steps modelled by levels five and six.

Step	Discovered invariants			All failures discharged?
	Glue	System	Total	
Level 2	0	1	**1**	Yes
Level 3	0	6	**6**	No
Level 4	0	4	**4**	Yes
Level 5	NA	NA	**NA**	NA
Level 6	NA	NA	**NA**	NA

NA = Not Applicable

Table 4.8: Summary of results for the Flash file development.

The failures that were not addressed at level 2 required the following invariants:

$$\forall r \cdot r \in dom(trans\_func) \Rightarrow flash(r) = flash2(trans\_func(r)) \tag{4.10}$$

$$programmed\_pages2 = trans\_func[programmed\_pages] \tag{4.11}$$

$$dom(flash2) = programmed\_pages2 \tag{4.12}$$

These invariants describe the relationships between the physical and logical views of the content of pages in the flash device. Invariant (4.10) is invented by HR; however, HREMO does not choose it as a candidate invariant because it does not pass the filter imposed by heuristic FH2; i.e. the left- and right-hand sides of the invariants are not disjoint since variable *trans_func* appears on both sides. Invariants (4.11) and (4.12) are also invented by HR; however, both invariants are needed to discharge the same failure; therefore, they are not selected as candidate invariants since HREMO only works for failures that require no more than one invariant in order to be discharged.

4.2.3 Summary of results

Table 4.9 summarises the results of applying our approach to each of the Event-B models we used during the development and the evaluation stages. All the experiments were performed with only basic typing invariants included in the models, i.e. no gluing or system invariants were present in the models. Table 4.9 shows for each refinement step, the number of failed POs as well as the number of gluing and system invariants discovered through our approach. We also record the number of iterations involved in the invariant discovery process. Note that the rows marked with a cross symbol (✗) in the column *Model proved* show the refinement

steps for which not all failures were addressed.

Event-B model	Step	Failed POs	Invariants				Model proved	Run time (sec)	
			Automatically discovered						
			Glue	System	Total	Iteration			
Traffic light	L2	2	2	0	**2**	1	✓	43	
Vending machine (Arith)	L2	6	3	0	**3**	1	✓	86	Development
Vending machine (sets)	L2	6	3	0	**3**	1	✓	56	set
Cars on a bridge	L2	2	1	0	**1**	1	✓	298	
	L3	6	0	5	**5**	1	✓	256	
	L4	7	0	5	**5**	1	✓	293	
Location Access Controller	L2	0	NA	NA	**NA**	NA	✓	0	
	L3	2	0	2	**2**	1	✓	20	
	L4	3	0	2	**2**	1	✓	66	
		2	0	2	**2**	2	✓	54	
		3	0	3	**3**	3	✓	51	
	L5	3	1	2	**3**	1	✓	106	
		6	1	3	**4**	2	✓	320	
		8	0	6	**6**	3	✓	283	
		5	0	3	**3**	4	✓	239	
		4	0	3	**3**	5	✓	220	
		1	0	1	**1**	6	✓	201	
Mondex	L2	4	0	1	**1**	1	✓	45	
		3	0	0	**0**	0	✗	46	
	L3	3	0	0	**0**	1	✗	130	
	L4	5	5	0	**5**	1	✓	72	
		6	1	4	**5**	2	✓	84	
		1	0	1	**1**	3	✓	51	
	L5	3	0	3	**3**	1	✓	167	
		5	0	4	**4**	2	✓	541	
		4	0	2	**2**	3	✓	357	Evaluation
	L6	15	5	1	**6**	1	✓	588	set
		25	0	14	**14**	2	✓	556	
		14	0	8	**8**	3	✓	425	
		13	0	8	**8**	4	✓	457	
		10	0	5	**5**	5	✓	341	
		8	0	4	**4**	6	✓	232	
		6	0	3	**3**	7	✓	262	
		3	1	1	**2**	8	✓	190	
		3	1	0	**1**	9	✓	36	
	L7	15	2	0	**2**	1	✓	282	
		5	1	0	**1**	2	✓	299	
		5	0	0	**0**	3	✗	270	
	L8	2	0	0	**0**	1	✗	203	
	L9	14	10	0	**10**	1	✓	179	
Flash File System	L2	1	0	1	**1**	1	✓	22	
	L3	4	0	1	**1**	1	✗	62	
		3	0	2	**2**	2	✓	46	
		2	0	1	**1**	3	✓	47	
		4	0	1	**1**	4	✓	47	
		2	0	1	**1**	5	✓	43	
		1	0	0	**0**	6	✗	38	
	L4	4	0	3	**3**	1	✓	76	
		1	0	1	**1**	2	✓	57	
	L5	0	NA	NA	**NA**	NA	✓	0	
	L6	0	NA	NA	**NA**	NA	✓	0	

NA = Not Applicable

Table 4.9: Automatically discovered invariants.

The run time for each iteration is also provided in Table 4.9. The calculated time does not include the generation of the simulation trace and the time of applying filtering heuristics FH4 and FH5 as they are currently performed manually. As can be observe, the Mondex development produces the highest run times, in particular the first iteration at level 6. Mondex contains the largest state from the case studies carried out; this results in a larger amount of generated conjectures, which increases the time of the discovery process. A possibility to reduce the execution time is to limit the search to only the concepts appearing in the failed POs; i.e. do not include the rest of the state as background information for HR. Further experiments are required in order to test this possible improvement.

In Table 4.10 we compare our results with the actual invariants given in the literature for the models used in the development set. Other developments are not compared because they were developed by us. Note that the invariants associated with levels five and seven of the Mondex system are not given in the literature. While all automatically discovered invariants are subsets of the invariants given in the literature, it is important to highlight that the automatically discovered invariants were sufficient to prove most of the refinement steps we encountered during our experiments.

Event-B model	**Step**	**Given in Literature**			**Automatically discovered**		
		Glue	**System**	**Total**	**Glue**	**System**	**Total**
Cars on a bridge	Level 2	1	1	**2**	1	0	**1**
	Level 3	0	5	**5**	0	5	**5**
	Level 4	0	23	**23**	0	5	**5**
Location access controller	Level 2	0	0	**0**	NA	NA	**NA**
	Level 3	0	3	**3**	0	2	**2**
	Level 4	0	3	**3**	0	7	**7**
	Level 5	1	6	**7**	2	18	**20**
Mondex	Level 2	2	4	**6**	0	1	**1**
	Level 3	2	0	**2**	0	0	**0**
	Level 4	8	11	**19**	4	5	**9**
	Level 5	-	-	-	0	9	**9**
	Level 6	10	3	**13**	7	45	**52**
	Level 7	-	-	**-**	3	0	**3**
	Level 8	1	0	**1**	0	0	**0**
	Level 9	10	0	**10**	10	0	**10**
Flash file system	Level 2	0	2	**2**	0	1	**1**
	Level 3	0	9	**9**	0	6	**6**
	Level 4	0	5	**5**	0	4	**4**
	Level 5	0	2	**2**	NA	NA	**NA**
	Level 6	0	4	**4**	NA	NA	**NA**

Note that the automatically discovered invariants are sufficient to discharge all failed POs generated when the invariants are absent from the models shown in the table. Our hypothesis is that the rest of the invariants introduce new requirements; thus, their absence does not produce proof failures.

Table 4.10: Comparison between hand-crafted and automatically discovered invariants.

Although we have identified some limitations of our approach, as discussed in Sections 4.2.1.1 and 4.2.2, it can be observed from Tables 4.9 and 4.10 that the automatic discovery of invariants through HREMO has provided promising results:

- In most cases, the set of invariants discovered through HREMO helped discharge all failed POs required to prove the refinement steps.
- The set of gluing invariants found by HREMO in each refinement step was almost identical to the set of gluing invariants given in the literature – remember that the exception cases in the Mondex model, i.e. levels two, three and eight, cannot be handled by HR's PRs – which have not been tailored for formal modelling.

With respect to system invariants, it can be observed that the last refinement of the cars on a bridge system shows a big gap between the invariants given in the literature and those found automatically with HREMO – although all failures are addressed by the automatically discovered invariants. As mentioned previously, we believe that this difference can be explained by the introduction of new requirements, resulting in the need for extra properties within the model. In the case of the cars on a bridge model, the last refinement introduces the physical environment of the system; for instance, the actual number of cars on the bridge. Thus, the new invariants represent connections between the physical and logical environments. These connections are not modelled at the abstract level; therefore, the conditions expressed by them are new requirements within the model, which are not identifiable through our approach. Details of the Event-B model can be found in [3].

Moreover, HREMO has shown that it works better at generating small invariants rather than compound invariants. This is demonstrated at levels five and six of the location access controller and within the Mondex case study, where HREMO identified more system invariants than were given in the literature. Both refinement steps introduce a partition of sets. In Event-B a partition can be specified by a single invariant using the predicate *partition(s,* s_1*, ...* s_n*)*, where s_1, ... s_n partition the set s, or by multiple invariants defining the containment and disjointness relationships, i.e. $s = s_1 \cup ... \cup s_n$ and $s_i \cap s_j = \varnothing$. HREMO generated invariants of the second type; this is reflected in the number of automatically discovered invariants being greater than the number of invariants given in the literature.

4.3 Summary

In this chapter we have outlined our system, HREMO, and in particular described the implementation details of the configuration and filtering heuristics. The overall process embodied within the implementation of our technique involves: i) the extraction of the animation traces, domain information and failed POs from an Event-B development, ii) the transformation of

such information into HR domain knowledge; this requires the application of the configuration heuristics, and iii) the selection of the candidate invariants; this requires the application of the filtering heuristics.

We have also shown the results of using HREMO to automatically discover invariants of seven Event-B models – all the experiments were performed with only basic typing invariants included in the models. A total of 23 refinement steps were involved in the seven case studies, which resulted in: 15 steps for which all the invariants needed to overcome the failures were discovered, five steps for which not all failures were addressed and three steps for which the approach was not applicable – because there were no failures associated with the steps. In the cases where HREMO failed to discover the needed invariants we could observe that was mainly because the current PRs could not produce the type of relationships expressed by the missing invariants. Bearing in mind that the theory formation process performed by HR has not been tailored for formal modelling, we can already observe the potential of ATF and HR for the automatically discovery of invariants. Further work is needed in order to build a theory formation tool dedicated to formal methods, in the case of HREMO, that work can be seen as the addition of new PRs and/or conjecture making techniques tailored for the formal modelling context.

Chapter 5

Comparative Studies and Further Applications of ATF

Generating invariants from the analysis of animation traces is an approach analogous to that of the Daikon system [44]; however, while Daikon is tailored for programming languages here we focus on formal models. In this chapter, a comparative study of HREMO and Daikon is presented.

Moreover, in the previous two chapters the technique was presented through the use of the Event-B formalism; however, the approach can be applied to other formalisms. In this chapter, this claim is supported through the application of the invariant discovery technique to a model written in the Z formalism. Finally, some remarks are made about how the technique can be exploited to support other aspects of formal modelling.

5.1 Comparative study with the Daikon system

Daikon is a system for the dynamic detection of likely invariants of programs. Its mechanism works by observing the values computed in a program execution and deriving properties that are true over such execution.

Daikon detects invariants for different programming languages, namely C, C++, Eiffel, Java, and Perl, and it can also detect invariants from record-structure data sources, such as spreadsheet files. To detect likely invariants, it evaluates selected variables at specified points within the code; these are called *program points*. Program points are usually object and procedure entries and exits, and variables are *program variables*, which are those explicitly written in the program code, e.g. class instance variables, procedure parameters, return values; and *derived variables*, which are aggregates of program variables that may not explicitly appear on the program code but that are in the scope of a procedure and may be of interest. An example of a derived variable is $a[i]$, where a represents an array and i an integer.

Daikon contains a collection of invariant templates[1] that are evaluated against the execution traces in order to determine if the template corresponds to an invariant of the program. The analysis performed by Daikon consists of the following steps:

- the templates are instantiated using the set of selected variables;
- then, the instantiated invariant templates are evaluated against the execution values; and
- any instantiated template that does not hold for all the states of the execution is removed from the set of likely invariants.

The output set of invariants is then post-processed in order to remove redundant or uninteresting invariants. For instance, Daikon removes less general invariants and invariants that express properties only about constants, etc.

Daikon has two main components: an *instrumenter* and an *inference engine*. The instrumenter selects the variables and program points in the target program and adds instructions into the code in order to generate trace data. The inference engine reads the traces and applies the invariant detection technique explained above. The inference engine is written in Java and is independent of the instrumenter; a separate instrumenter is required for each programming language.

Daikon and HREMO have a number of similarities:

- both approaches depend on data traces to search for candidate invariants – ProB animation traces in our work and program test suites for Daikon;
- both contain an inference engine which is language independent;
- Daikon selects program and derived variables as "objects of interest" for which invariants are searched. This is equivalent to the selection of core and non-core concepts in our approach, where core concepts relate to program variables and non-core concepts relate to derived variables; and
- both share common invariant elimination strategies such as the elimination of less general invariants.

However, the approaches differ in that while Daikon selects the invariants from a set of invariant templates HREMO uses general purpose production rules to generate conjectures about the domain. Also, while Daikon selects program and derived variables from the code, HREMO selects the core and non-core concepts from the failed POs. Moreover, Daikon not only produces global invariants, but also invariants that represent pre- and/or post-conditions of specific procedures within the code, while HREMO only generates global invariants, which hold before and after the execution of all procedures, i.e. events, in the model. Finally, it should also be stressed that Daikon is a system designed with program analysis in mind, whereas the work presented here is an initial investigation into developing an invariant gen-

[1] In [44] it was reported that Daikon contained 75 invariant templates and 25 derived variable templates.

eration tool for refinement based formal methods.

5.1.1 Experiments

Two experiments were carried out in order to compare Daikon with HREMO. The experiments consisted of writing a Java program which is equivalent to the first and second refinements of Abrial's "cars on a bridge" model [3] and in comparing the invariants detected by Daikon with the invariants detected by HREMO for each refinement. This model was selected because it provides a set of invariants that facilitated the comparison, i.e. invariants that fit the templates of Daikon.

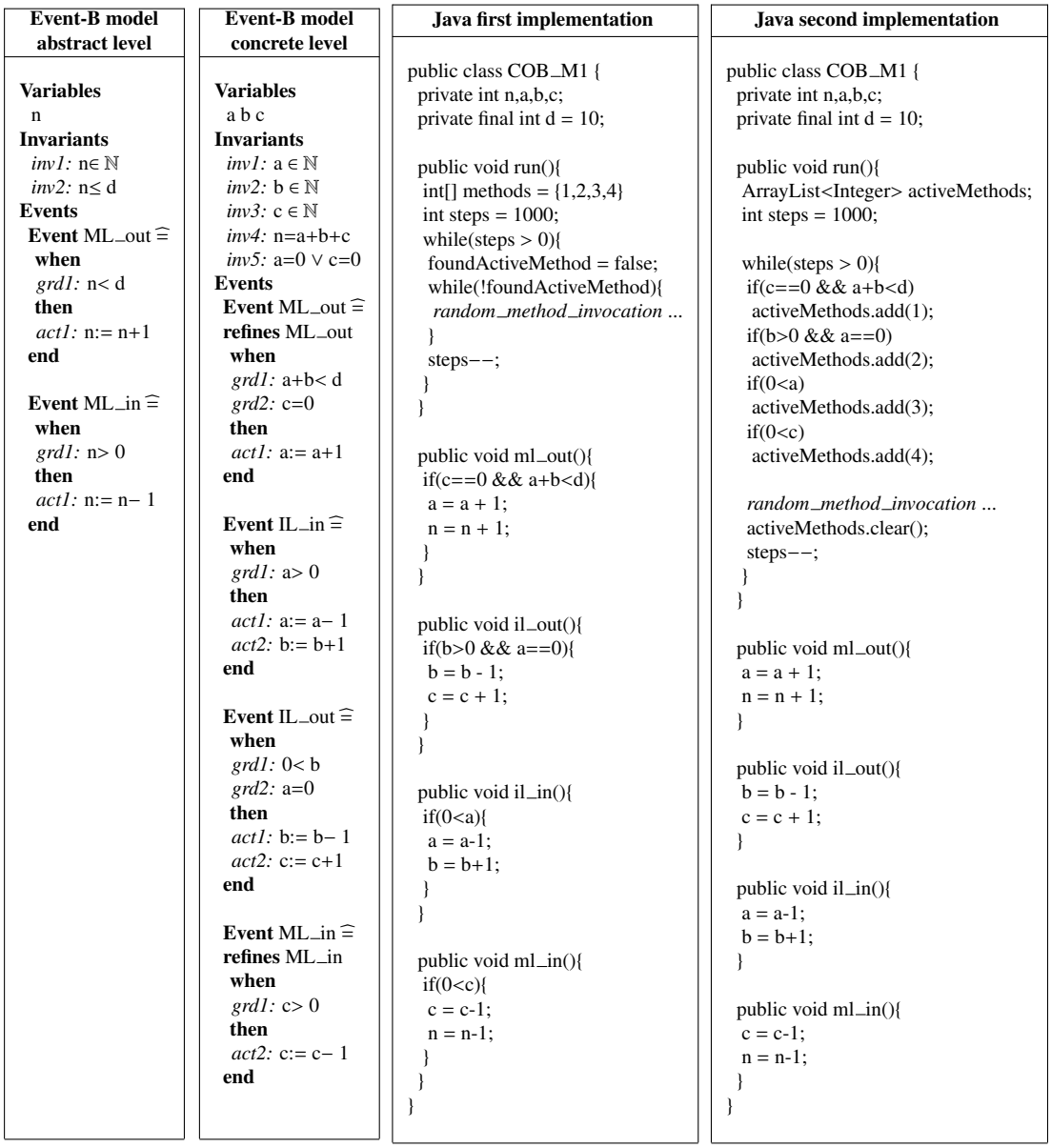
The transformation from the Event-B model to Java consisted of converting:

- a refinement step, i.e. an abstract and a concrete level, into one Java class which merges the behaviour from both levels;
- variables and constants into instance variables and constants of a Java class;
- each event into a Java method;
- the guards of an event into conditional statements; and
- the actions into instructions to be executed by the correspondent method.

In addition, a method *run* was introduced into the Java code in order to simulate the random execution of methods that occurs within ProB.

Experiment 1: First refinement. The "cars on a bridge" system models the control of car traffic on a single lane bridge that connects a mainland to an island. In the first refinement step, at the abstract level cars are modelled leaving and entering the island while at the concrete level the requirement that the bridge only supports one way traffic is introduced. Figure 5.1 shows the Event-B specification and two Java implementations of this refinement step where, n represents the total number of cars at the abstract level while at the concrete level a, b and c represent the cars going from the mainland to the island, the cars on the island and the cars going from the island to the mainland, respectively.

The two Java implementations in Figure 5.1 differ from each other in the placement of the guards inside the Java code. In the first implementation the guards are conditionals located inside the correspondent Java method, i.e. after the method triggers the conditions are checked; while in the second implementation the guards are conditionals placed within the *run* method, i.e. first the conditions are checked and then, if the conditions are true, the corresponding method is executed.



Event-B model abstract level

Variables
n
Invariants
inv1: n∈ ℕ
inv2: n≤ d
Events
Event ML_out ≙
when
grd1: n< d
then
act1: n:= n+1
end

Event ML_in ≙
when
grd1: n> 0
then
act1: n:= n− 1
end

Event-B model concrete level

Variables
a b c
Invariants
inv1: a ∈ ℕ
inv2: b ∈ ℕ
inv3: c ∈ ℕ
inv4: n=a+b+c
inv5: a=0 ∨ c=0
Events
Event ML_out ≙
refines ML_out
when
grd1: a+b< d
grd2: c=0
then
act1: a:= a+1
end

Event IL_in ≙
when
grd1: a> 0
then
act1: a:= a− 1
act2: b:= b+1
end

Event IL_out ≙
when
grd1: 0< b
grd2: a=0
then
act1: b:= b− 1
act2: c:= c+1
end

Event ML_in ≙
refines ML_in
when
grd1: c> 0
then
act2: c:= c− 1
end

Java first implementation

```
public class COB_M1 {
 private int n,a,b,c;
 private final int d = 10;

 public void run(){
  int[] methods = {1,2,3,4}
  int steps = 1000;
  while(steps > 0){
   foundActiveMethod = false;
   while(!foundActiveMethod){
    random_method_invocation ...
   }
   steps−−;
  }
 }

 public void ml_out(){
  if(c==0 && a+b<d){
   a = a + 1;
   n = n + 1;
  }
 }

 public void il_out(){
  if(b>0 && a==0){
   b = b - 1;
   c = c + 1;
  }
 }

 public void il_in(){
  if(0<a){
   a = a-1;
   b = b+1;
  }
 }

 public void ml_in(){
  if(0<c){
   c = c-1;
   n = n-1;
  }
 }
}
```

Java second implementation

```
public class COB_M1 {
 private int n,a,b,c;
 private final int d = 10;

 public void run(){
  ArrayList<Integer> activeMethods;
  int steps = 1000;

  while(steps > 0){
   if(c==0 && a+b<d)
    activeMethods.add(1);
   if(b>0 && a==0)
    activeMethods.add(2);
   if(0<a)
    activeMethods.add(3);
   if(0<c)
    activeMethods.add(4);

   random_method_invocation ...
   activeMethods.clear();
   steps−−;
  }
 }

 public void ml_out(){
  a = a + 1;
  n = n + 1;
 }

 public void il_out(){
  b = b - 1;
  c = c + 1;
 }

 public void il_in(){
  a = a-1;
  b = b+1;
 }

 public void ml_in(){
  c = c-1;
  n = n-1;
 }
}
```

Figure 5.1: Event-B and Java models of the cars on a bridge system.

Table 5.1 shows the results of the invariant analysis from both approaches for the models shown in Figure 5.1. The first column lists the invariants at the concrete level of the model – since the invariant analysis is intended for the concrete level, the invariants at the abstract level are ignored – the second column lists the invariants detected by HREMO, and columns three and four lists some of the invariants detected by Daikon for the two Java implementations respectively. The output from Daikon is too large and for this reason we only show a fragment of it, that is: the object invariants and the invariants for the method *ml_out* at entry and exit points.

The first three invariants of the model, i.e. $a \in \mathbb{N}$, $b \in \mathbb{N}$ and $c \in \mathbb{N}$ are type invariants.

Invariants of the model	HREMO Reported Invariants	Daikon Reported Invariants First Implementation	Daikon Reported Invariants Second Implementation
$a \in \mathbb{N}$	n = a + b + c	==========	==========
$b \in \mathbb{N}$		cob.COB_M1:::OBJECT	cob.COB_M1:::OBJECT
$c \in \mathbb{N}$		this.n >= 0	this.n >= 0
$a = 0 \vee c = 0$		this.a >= 0	this.a >= 0
n = a + b + c		this.b >= 0	this.b >= 0
		this.c >= 0	this.c >= 0
		this.d == 10	this.d == 10
		this.n >= this.a	this.n >= this.a
		this.n >= this.b	this.n >= this.b
		this.n >= this.c	this.n >= this.c
		this.n <= this.d	this.n <= this.d
		this.a < this.d	this.a < this.d
		this.b <= this.d	this.b <= this.d
		this.c < this.d	this.c < this.d
		==========	==========
		cob.COB_M1.ml_out():::ENTER	cob.COB_M1.ml_out():::ENTER
		==========	this.c == 0
		cob.COB_M1.ml_out():::EXIT	this.n - this.a - this.b == 0
		this.b == orig(this.b)	==========
		this.c == orig(this.c)	cob.COB_M1.ml_out():::EXIT
		this.d == orig(this.d)	this.c == 0
		this.n >= 1	this.n >= 1
		this.n >= orig(this.n)	this.a >= 1
		this.n > orig(this.a)	this.n - orig(this.n) - 1 == 0
		this.a >= orig(this.a)	this.a - orig(this.a) - 1 == 0
		this.b <= orig(this.n)	this.n - this.a - this.b == 0
		this.c <= orig(this.n)	this.n - this.b - orig(this.a) - 1 == 0
		this.d >= orig(this.n)	this.a + this.b - orig(this.n) - 1 == 0
		this.d > orig(this.a)	this.b - orig(this.n) + orig(this.a) == 0
		...	...

Table 5.1: Comparison of expected and detected invariants for the first refinement of the "Cars on a bridge" model.

While Daikon reports at the object level that: $a \geq 0$, $b \geq 0$ and $c \geq 0$, HREMO does not deal with these type of invariants. This is because, although HR generates these conjectures from the domain, type invariants are always supplied by the user and therefore, HREMO does not report them as candidate invariants. Regarding invariant $a=0 \vee c=0$, neither HREMO nor Daikon report it in their outputs. In the case of HREMO the reason for this is that $a=0 \vee c=0$ represents a system invariant, i.e. introduces a new requirement to the model. For the model of cars on a bridge the absence of this invariant does not produce any failed PO; therefore, if no failed POs are associated with the absence of an invariant HREMO fails to identify and report it. Finally, the gluing invariant $n = a+b+c$ is reported by HREMO but not by Daikon.

From this experiment it was also noted that different Java implementations of the same behaviour yielded different likely invariants from Daikon. As can be observed in Table 5.1, the outputs from Daikon for the entry and exit points of the method *ml_out* in the two Java implementations are different. In the second implementation, which uses method calls after the conditions have been checked, Daikon reports a fragment of the gluing invariant for the entry and exit points of the method *ml_out*, namely: $n - a - b == 0$ or $n = a + b$.

Experiment 2: Second refinement. In this refinement step traffic lights are introduced into the model. The Event-B and Java developments are not presented here since they are similar to the ones presented for the first refinement; however, details of the Event-B development can be found in [3]. In terms of translating Event-B into Java, the same approach as described for experiment 1 was followed. The results from this experiment are shown in Table 5.2. The first four invariants of the model are type invariants, while the other seven specify new requirements due to the addition of the traffic lights. HREMO reported five of the seven system invariants while Daikon did not report any. From the output of the second implementation it can again be observed that for the entry and exit points of method *ml_out* Daikon produces fragments of the invariants which were not produced in the first implementation, e.g. the colour of the traffic lights, i.e. *cob.Color.RED == this.il_tl* and *cob.Color.GREEN == this.ml_tl* and the number of cars going from the island to the mainland, i.e. *c == 0*; nevertheless Daikon does not produce global relationships between these values; for instance that *ml_tl=green ⇒ c=0*, which states that whenever the mainland traffic light is green, there are no cars travelling in the opposite direction.

Invariants of the model	**HREMO Reported Invariants**	**Daikon Reported Invariants First Implementation**	**Daikon Reported Invariants Second Implementation**
ml_tl ∈ { red,green}	ml_tl = green ⇒ c = 0	==========	==========
il_tl ∈ { red,green}	il_tl = green ⇒ a = 0	cob.COB_M2:::OBJECT	cob.COB_M2:::OBJECT
il_pass ∈ { 0,1}	ml_tl = red ⇒ ml_pass = 1	this.a >= 0	this.a >= 0
ml_pass ∈ { 0,1}	il_tl = red ⇒ il_pass = 1	this.b >= 0	this.b >= 0
ml_tl = green ⇒ c = 0	il_tl = red ∨ ml_tl = red	this.c >= 0	this.c >= 0
ml_tl = green ⇒ a+b < d		this.d == 5	this.ml_tl != null
il_tl = green ⇒ a = 0		this.ml_tl != null	cob.Color.GREEN != null
il_tl = green ⇒ b > 0		cob.Color.RED != null	cob.Color.RED != null
ml_tl = red ⇒ ml_pass = 1		cob.Color.GREEN != null	this.il_tl != null
il_tl = red ⇒ il_pass = 1		this.il_tl != null	this.a < this.d
il_tl = red ∨ ml_tl = red		this.a <= this.d	this.b <= this.d
		this.b <= this.d	this.c < this.d
		this.c <= this.d	==========
		==========	cob.COB_M2.ml_out():::ENTER
		cob.COB_M2.ml_out():::ENTER	this.ml_tl == cob.Color.GREEN
		this.c < this.d	cob.Color.RED == this.il_tl
		==========	this.c == 0
		cob.COB_M2.ml_out():::EXIT	this.il_pass == true
		this.b == orig(this.b)	this.a >= this.c
		this.c == orig(this.c)	this.b >= this.c
		this.d == orig(this.d)	this.b < this.d
		this.il_tl == orig(this.il_tl)	==========
		this.il_pass == orig(this.il_pass)	cob.COB_M2.ml_out():::EXIT
		this.ml_pass == true	cob.Color.GREEN == orig(this.ml_tl)
		this.a >= orig(this.a)	cob.Color.RED == orig(this.il_tl)
		this.c < this.d	this.il_pass == this.ml_pass
		this.d >= orig(this.a)	this.a >= 1
		...	this.c == 0
			this.il_pass == true
			...

Table 5.2: Comparison of expected and detected invariants for the second refinement of the "Cars on a bridge" model.

Daikon provides a large set of options and filters to control the processing and output of the likely program invariants. For instance, for each invariant template there is a configuration enable switch that can be enabled or disabled. The type of derived variables that should be involved in the discovery process can also be controlled through Daikon configuration options. Filters to limit the invariants that are reported are also available. Because of the vast number of available options it is difficult to find the optimal settings with which to run Daikon. Daikon was run with its default setting over the Java programs for both refinements of the cars on a bridge model but none of the system and gluing invariants were detected and no additional invariant templates (apart from those enabled by default) seem to fit the nature of the model. Furthermore, reading the Daikon documentation suggested that for the second experiment a *splitter file* was required. This is a configuration file needed in order to create conditional invariants. The splitter file can be manually supplied or automatically created through one of Daikon command line instructions. However, running Daikon using the splitter file did not generate the expected invariants either. Because of the common structure and not high complexity of the invariants of the cars on a bridge model, we believe that given sufficient knowledge of the configuration options, Daikon can be properly configured resulting in the detection of these invariants; however, expert knowledge about the discovery mechanism is required. Like Daikon, HREMO is configurable; nevertheless, the configuration performed with our technique is completely automatic and it does not require the user to have any knowledge about how the detection process works.

From the results of the experiments presented above it can be concluded that:

1. Different implementations of the same behaviour affect the analysis performed by Daikon, resulting in the generation of different invariants for each implementation. Furthermore, it seems that the use of method or function calls to perform tasks of the system is crucial in order to obtain better results from Daikon.

2. Daikon performs very well at finding pre- and post-conditions of methods; however, based on our experiments, Daikon has difficulty at detecting global invariants.

3. HREMO performs better than Daikon at finding global invariants and in particular at finding gluing invariants. However, system invariants represent a challenge for HREMO when there is no proof failure associated with the absence of the invariant.

4. HREMO only detects global invariants; pre- and post-conditions are not part of its intended output.

5. Daikon is restricted to the available invariant templates, and although it is possible to extend Daikon to add new templates, this means that possible interesting invariants may be missed by the inference process because there is no a template available.

HREMO is not restricted by patterns of conjectures but it is restricted by the general purpose PRs provided by HR; however, the iterative application of the PRs provides greater flexibility in terms of the kinds of invariants that can be discovered.

5.2 Application to other formalisms

As mentioned previously, we believe that the invariant discovery approach presented in this thesis can be applied beyond Event-B. To illustrate this, a Z specification of a vending machine system is used in order to show the mapping between the data from the model and the required components of HREMO. The example, which is taken from [118], is shown in Figure 5.2. The specification models a vending machine which dispenses drinks when a three-digit code is typed by a customer. In the abstract model, the action of entering the code to the machine is modelled by an atomic step, i.e. the operation *Choose*, while in the concrete model the three digits of the code are input one-by-one, i.e. operations *FirstPunch* and *NextPunch*.

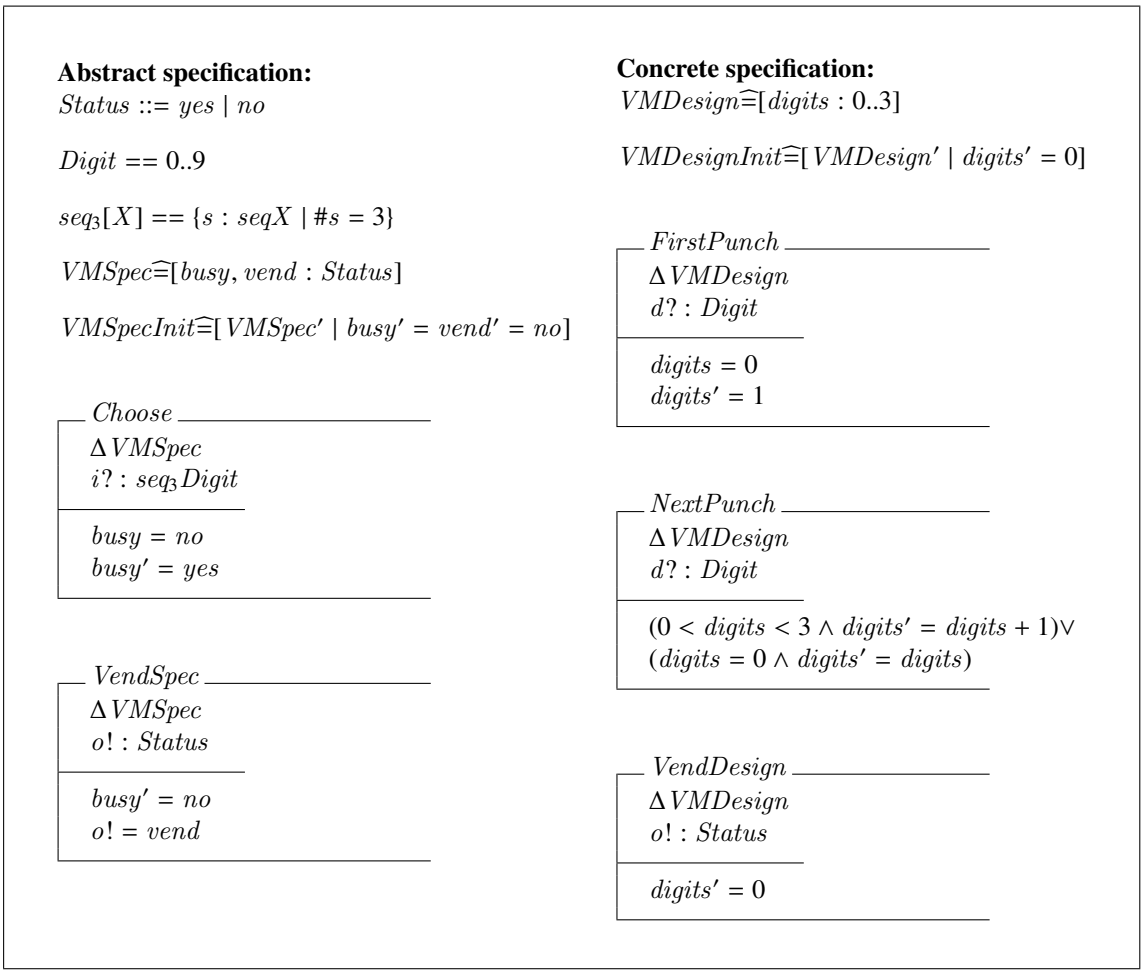


Figure 5.2: Z specification of a vending machine system taken from [118]

The abstract model defines the types *Status* (to track progress of a transaction), *Digit* (possible digits in the code), *seq$_3$[X]* (set of sequences with length 3), *busy* (to specify if the vending machine is in use) and *vend* (to specify if a transaction was successful), while the concrete model introduces the type *digits* (to record the number of digits entered). The schemas *VendSpec* and *VendDesign* model the end of a transaction in the abstract and concrete models respectively, their outputs indicate if the transaction was successful or not. From this specification, the following core concepts are identified:

$$\textit{concepts T1} = \{\textit{Status}, \textit{Digit}\}$$
$$\textit{concepts T2} = \{\textit{seq}_3[\mathrm{X}], \textit{busy}, \textit{vend}, \textit{digits}\}$$

where the examples of *Status* and *Digit* are {*yes*, *no*} and {0,1,2,3,4,5,6,7,8,9}, respectively. The examples required for the T2 concepts can be obtained from an animator for Z; for instance, the ProZ animator [106]. In order to prove that the concrete schema *FirstPunch* refines the abstract schema *Choose*, the following PO must be proved:

$$\forall\, busy, vend : Status;\ digits, digits' : 0..3;\ seq_3 Digit;\ d? : Digit \bullet$$
$$busy = no \land digits = 0 \land digits' = 1 \Rightarrow \exists\, busy', vend' : Status \bullet busy' = yes$$

As it stands, the PO is not provable, a retrieve relation that explains the correspondence between the abstract and concrete models is required. HR$_{\text{EMO}}$ can be used to find the retrieve relation. In order to do so, the prioritised core and non-core concepts must be identified from the failed PO, i.e.

core concepts = {*busy, vend, digits, seq_3[X], Digit*}
non-core concepts = {*busy=no, digits=0, digits=1, busy=yes,*
(digits = 0 ∧ digits = 1),
(busy = no ∧ digits = 0 ∧ digits = 1) }

The first four non-core concepts can be represented in HR$_{\text{EMO}}$ by applying the *split* PR to core concepts *busy* and *digits* where the split values are *no, 0, 1* and *yes*. The remaining non-core concepts can be obtained through the application of the *compose* PR. Furthermore, the following PRs are selected:

PRs = {*compose, disjoint, negate, split*}

where the *compose, disjoint and negate* PRs are always enabled by default while the *split* PR is enabled because of the identified non-core concepts. The required retrieve relation is:

$$busy = no \Leftrightarrow digits = 0$$

which is a type of relationship previously shown to be identifiable by HR$_{\text{EMO}}$.

Through the Z example it has been illustrated how HREMO could be applied to a different formalism. The key features of this broader application are: firstly, the ability to map data types and operators within the given formalism onto core concepts and PRs within HREMO; secondly, the formalism must be supported by simulation and formal verification.

5.3 Beyond current applications

5.3.1 Debugging models

The experiments carried out in the previous chapter revealed an inconsistency between levels five and six of the re-constructed Mondex system model. In the last iteration of invariant discovery, HREMO failed to find conjectures to discharge two failed POs. At this point the invariant discovery process is not longer applicable since no failures are addressed by the iteration. This may occur because:

1. HREMO cannot generate the missing invariant, or
2. more than one invariant is needed to discharge the same failure, or
3. the refinement step is not correct; either because there is an error in the invariants or in the model.

The failed POs generated by the model at this iteration are shown in Figure 5.3. These POs represent failures in the preservation of one of the invariants discovered by HREMO, that is:

$$idleT = currentT[idleTP]$$

This is a gluing invariant used to express that the state set *idleT* is replaced in the concrete model by the image of function *currentT* over the set *idleTP*.

idleT = currentT[idleTP]
p1 ∈ idleFP
p2 ∈ abortepvP
p1 ↦ t ∈ currentF
p2 ↦ t ∈ currentT
from(t) = p1
to(t) = p2
⊢
idleT = ({p2} ⩤ currentT)[idleTP]

(a) Failed PO1

idleT = currentT[idleTP]
p1 ∈ aborteprP
p2 ∈ idleTP
p1 ↦ t ∈ currentF
p2 ↦ t ∈ currentT
from(t) = p1
to(t) = p2
⊢
idleT = ({p2} ⩤ currentT)[idleTP \ {p2}]

(b) Failed PO2

Figure 5.3: Level 6 Iteration 10: Failed POs.

The failed POs shown in Figure 5.3 are in fact unprovable. As can be observed, the equality expressed by the invariant holds in the hypothesis of the POs; however, the right-hand side

of the equality is modified in the goals by removing the element *p2* from the domain of function *currentT* while the left-hand side remains unmodified. The events associated with the failed POs are new events introduced in the concrete model. Their role is to register the end of a transaction when either the source or the target purse has failed while the other side of the transaction is still in the idle state. The failed POs suggest that:

1. there is an inconsistency between the behaviour of the concrete and abstract models, or

2. there is an error in the invariant.

In order to solve these failures either an abstract version of the events should be modelled in level five of the development, or the invariant at level six should be removed or modified. By manual inspection of the model we realised that in level five we had not handled all possible cases for which a transaction could end because of an abort state. The inconsistency was solved by adding new events in level five that handled all possible states in which a transaction could fail.

Note that this analysis is not currently performed by HREMO. However, through the automatically discovered invariants the inconsistency in the model was revealed. These failures are not generated in the original development of the Mondex system presented in [22], because the gluing invariant is defined as:

$$currentT[idleTP] \subseteq idleT$$

which expresses a weaker relationship than the invariant suggested by HREMO.

5.3.2 Handling incorrect models

Our approach to invariant discovery assumes that the models are correct. Having the ability of identifying incorrect models would increase the effectiveness of HREMO. A technique that aims at providing modelling guidance when a failed refinement occurs, has been developed in this thesis. This technique is called *Refinement Plans* [55]. Refinement plans are used to identify if a known pattern of refinement is closely aligned to the given failed refinement. When a match is found, common patterns of failure at the level of POs and models are analysed in order to provide modelling guidance as to how to overcome the failure; for instance, the introduction of missing invariants. However, when a common pattern of failure is instantiated by a particular refinement step, the associated guidance will typically only be partially instantiated. In such cases, for instance when the guidance is a partially instantiated invariant schema, HREMO can be used to complete its instantiation. The next chapter presents the refinement plans approach and shows how refinement plans and HREMO complement each

other by using the partially instantiated guidance obtained from the plans to tailor the search for invariants in HREMO and provide complete solutions to the user.

5.4 Summary

This chapter presented a comparative case study between HREMO and Daikon, a dynamic invariant detector that uses program executions to instantiate invariant templates in order to find likely invariants. The case study consisted on the analysis of two Java programs equivalent to the first and second refinements of the Event-B model of the cars on a bridge system. Daikon was successful at finding typing invariants, and pre- and post-conditions of each method in the program; however, it failed to find the global invariants associated with the model. On the other hand, HREMO discovered 6 out of the 9 global invariants but neither typing invariants nor pre- and post-conditions were reported since they are not part of its intended output. Although both systems are configurable, HREMO configuration is performed automatically while Daikon must be configured by the user–this involves enabling invariant templates, generation of special files that enable the search of conditional invariants, etc. Given the right configuration we believe that the invariants of the model can be generated by Daikon; however, this requires the intervention of an expert user of the system.

The generality of the invariant discovery approach presented in this thesis was also illustrated by explaining how the mapping of a Z specification would be performed by HREMO. This showed that as long as a formalism is supported by simulation tools and the reasoning is underpinned by proof, HREMO can be used to discover invariants of its models.

Finally, it has been discussed how HREMO can provide a deeper insight about the model by making more evident inconsistencies that otherwise the developer may not notice. For instance, the invariants found by HREMO revealed the absence of some events in the reconstructed model of the Mondex system. Such situations also motivate the need for handling models that are not correct. Refinement plans have been developed in this thesis for this purpose. The next chapter introduces this technique and shows how the integration between top-down approaches, such as refinement plans, with bottom-up approaches, such as HREMO, can provide greater flexibility and increase effectiveness of both strategies of design.

Chapter 6

Modelling Guidance with Refinement Plans

Many approaches to design, whether informal [48] or formal [6, 21, 46, 47], rely on patterns. This is also the case for refinement plans, a technique that combines common patterns of refinement with common patterns of failure in order to provide automatic modelling guidance.

In this chapter a classification of refinement patterns is described which have been identified from the analysis of Event-B developments. Moreover, the mechanism for refinement plans is introduced and developed for one of the identified patterns in the hierarchy. Finally, an integration of refinement plans and HREMO is explored. That is, the use of a refinement plan instantiation (or partial instantiation) to tailor HREMO . Exploiting the synergy between the approaches results in an improvement of HREMO's mechanism for the discovery of invariants as well as providing flexibility for the patterns handled by the refinement plans.

6.1 Approach

Consider the Event-B model[1] shown in Figure 6.1. At the abstract level, the model shows event *incr*, which increments variable *x* by *y* in an atomic step:

$$x := x + y.$$

At the concrete level the value of *y* is assigned one unit at a time into a temporary accumulator variable *x_tmp* in event *step*:

$$x\_tmp := x\_tmp + 1.$$

This iterative process is controlled using variable *n* which is also incremented by event

[1] This example has been extracted from [39].

step and guarded by condition:

$$n < y.$$

When the iterative process terminates, in other words, when

$$n = y$$

the value contained by the accumulator variable *x_tmp* is assigned to variable x in event *end_ok*, which in turn refines event *incr*. The new event, *start*, initialises the variables so that the iterative process carried out by event *step* can be enabled. The invariant:

$$\mathit{flag} = \mathit{FALSE} \Rightarrow \mathit{x\_tmp} = x + n$$

specifies that when the iterative process starts the value of variable *x_tmp* is always equal to the accumulated value; i.e. *x+n*. Using HREMO to find this invariant does not produce positive results after the default 1000 theory formation steps. This does not imply that the invariant cannot be found, rather it means that additional search is required. However, as will be shown in Section 6.3, patterns of refinement provide additional information that helps further tailor HREMO, leading to the discovery of the invariant.

ABSTRACT MODEL:

Variables
x y
Invariants
x ∈ ℕ
y ∈ ℕ

Event incr ≙
then
x := x+y
end

CONCRETE MODEL:

Variables
x y n x_tmp flag
Invariants
n ∈ ℕ
x_tmp ∈ ℕ
flag ∈ BOOL
flag=FALSE ⇒ x_tmp=x+n

Event start ≙
when
flag = TRUE
then
n := 0
x_tmp := x
flag := FALSE
end

Event step ≙
when
n < y
flag = FALSE
then
x_tmp := x_tmp+1
n := n+1
end

Event end_ok ≙
refines incr
when
flag = FALSE
n = y
then
x := x_tmp
flag := TRUE
end

Figure 6.1: Addition example [39].

This type of refinement, in which an action is executed in an atomic step at the abstract level and refined at the concrete level via iteration, represents a common pattern of refine-

ment. Here we call it *the accumulator pattern*[2]. More specifically, the accumulator pattern allows the modelling of protocols that require to gradually process data of the system before a conclusive action can take place.

As illustrated in Section 2.2, Event-B refinements contain a *modelling* and a *reasoning* component. The latter takes the form of POs that must be discharged. Here, a pattern of refinement is described in terms of its modelling and PO patterns, which are instantiated with the user's development. Refinement plans use the identified instantiations in order to i) classify refinement steps in a development, and ii) provide modelling guidance when a development is flawed or is hard to verify but close to a known pattern. By combining knowledge from PO failures with knowledge about the user intentions, in the form of partial pattern instantiations, refinement plans automatically generate modelling alternatives that help overcome the failures. We use a schematic representation in order to describe the key elements of the modelling and PO patterns associated with the accumulator pattern. The schemas are shown in Figures 6.2 and 6.3.

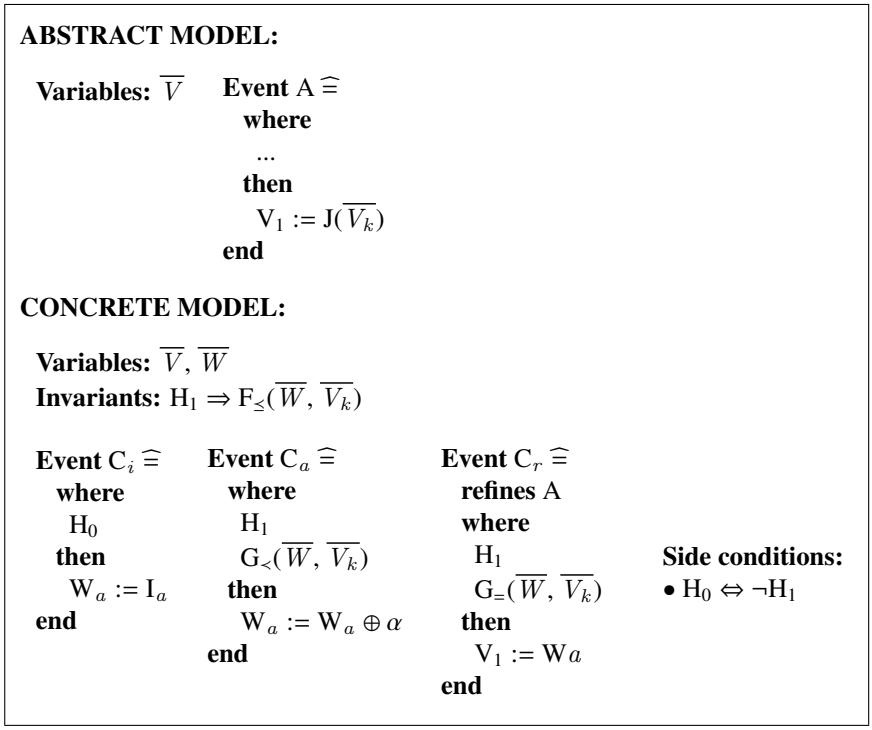


Figure 6.2: Accumulator pattern – Modelling schema.

where:

- Vs and Ws denote abstract and concrete meta-variables, respectively.
- $\overline{V}$ and $\overline{W}$ denote lists of meta-variables.

[2]This pattern has been inspired by the work presented in [22] where a technique for breaking up the atomicity of an event was introduced by Butler and Yadav and further developed in [21, 46, 47].

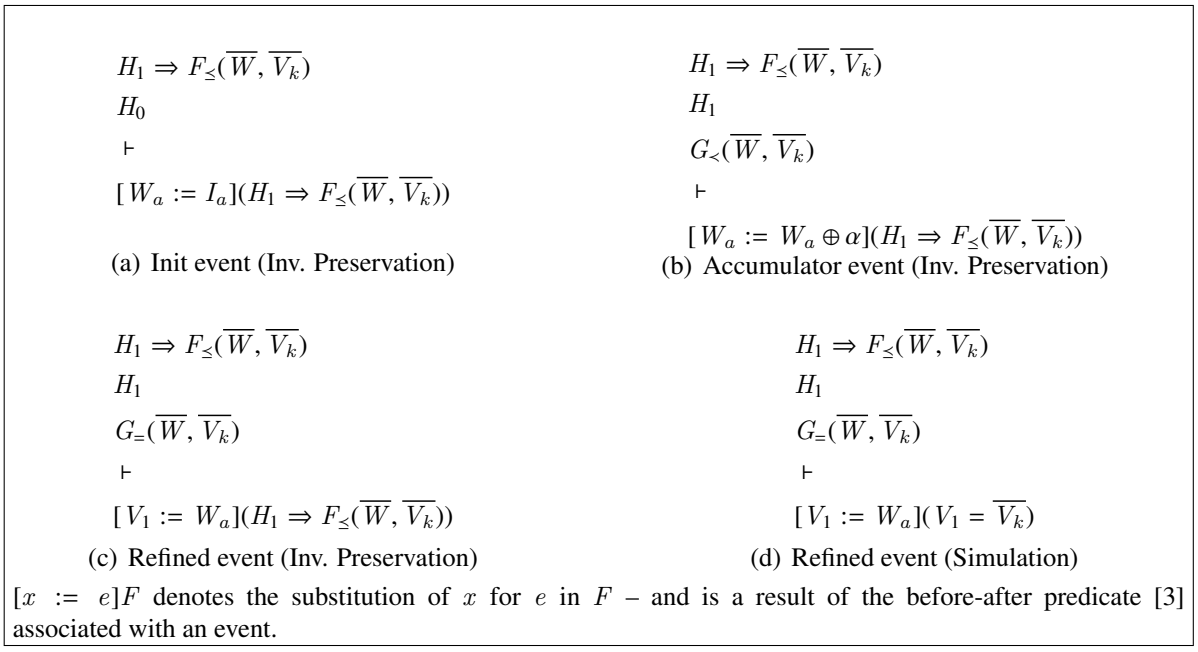


Figure 6.3: Accumulator plan – POs schema.

- I and J represent expressions used in assignments to meta-variables.
- F, G and H denote meta-predicates, where subscripts are used to restrict their instantiation, e.g. $G_=$ restricts G to be an equality.
- α denotes a meta-term that can be instantiated with any suitable basic data type value or Event-B element, e.g. constant, variable, event parameter, etc.
- $\oplus$ denotes an additive operation (either for the natural numbers or for sets).

Thus, based on these schemas, the key elements of the accumulator pattern are:

- The abstract model has an event (A) that is refined in the concrete model.
- A set of new variables $\overline{W}$ = {W_1, ..., W_n} are introduced.
- A variable $W_a \in \overline{W}$, which denotes an accumulator variable. That is, W_a has an associated initialisation, accumulator and refined event, i.e. events with the action patterns W_a:=I_a, W_a:=$W_a \oplus \alpha$ and V_j:=W_a, respectively – where I_a represents the initial value assigned to meta-variable W_a.
- The initialisation event (C_i), accumulator event (C_a), and refined event (C_r).
- An invariant, $H_1 \Rightarrow F_{\leq}(\overline{W}, \overline{V_k})$, that explains the refinement, i.e. that the content of the accumulator variable is contained within the value assigned in the abstract model – the $\leq$ symbol generalises the containment relationship.
- The initialisation, accumulator(s), and refined events must preserve the invariant, Figures 6.3(a), 6.3(b) and 6.3(c), respectively.
- The refined event must simulate the abstract action, Figure 6.3(d).

Instantiating the key elements listed above with the example presented in Figure 6.1 yields the following:

- Abstract event: *incr*.
- Refined event: *end_ok*.
- Set of new variables $\overline{W}$={*n, x_tmp, flag*}.
- Accumulator variable W_a=*x_tmp* – since the action patterns $W_a := I_a$, $W_a := W_a \oplus \alpha$, and $V_j := W_a$ occur in events *start, step* and *end_ok*; i.e. actions *x_tmp:=x*, *x_tmp:=x_tmp+1* and *x:=x_tmp*, respectively.
- Initialisation event C_i = *start*, accumulator event C_a = *step*, and refined event C_r = *end_ok*.
- Invariant *flag=FALSE* $\Rightarrow$ *x_tmp=x+n*, since it fits the pattern $H_1 \Rightarrow F_{\leq}(\overline{W}, \overline{V_k})$ where the containment relationship $\leq$ is instantiated with =.
- The model preserves the invariant and simulates the abstract action.

6.1.1 Roles of refinement plans

Refinement plans are heuristic in nature, and can be applied *flexibly* during a development. This flexibility is achieved through *partial matching*, *proof-failure analysis*, and the use of *theory formation*, in particular the HRemo system. Moreover, we envisage a variety of scenarios in which refinement plans can play a role within the development of formal models:

Correcting refinements: a flawed refinement step may be overcome by modifications to the abstract and/or concrete models, e.g. guard strengthening, invariant discovery.

Layering refinements: an overly complex refinement step can give rise to unproven proof obligations – such failures may be overcome by the introduction of intermediate layers of abstraction.

Abstracting refinements: a development which starts at a too concrete level may benefit from guidance as to how to reduce the initial complexity and open up alternative modelling choices.

Suggesting refinements: at the fringe of a development, suggesting alternative refinement steps could be beneficial to a users productivity.

Increasing proof automation: our refinement plans will enable us to exploit the correspondence between the structure of a refinement and the pattern of proof associated with its verification.

So far the focus has been in the application of refinement plans for the first role, i.e. correcting refinements. In particular, only failures at the concrete level have been explored; i.e. where the abstract level is assumed to be correct. However, the exploration of other roles is planned for future work. Further details are given in Chapter 8.

6.1.2 Refinement patterns

The hierarchy of refinement patterns shown in Figure 6.4 was identified through the analysis of a set of Event-B developments taken from the literature. Details of these developments can be found in Chapter 7 (Section 7.2). All nodes denote a distinct pattern of refinement. Moreover, a development may contain an instance of a pattern only if the pattern immediately above in the hierarchy is also identified in the development. This reflects the sharing of properties between patterns within the hierarchy. In other words, the conditions of the ancestor patterns are also conditions for the descendants. All the patterns are briefly described below and their details are given in Appendix A.

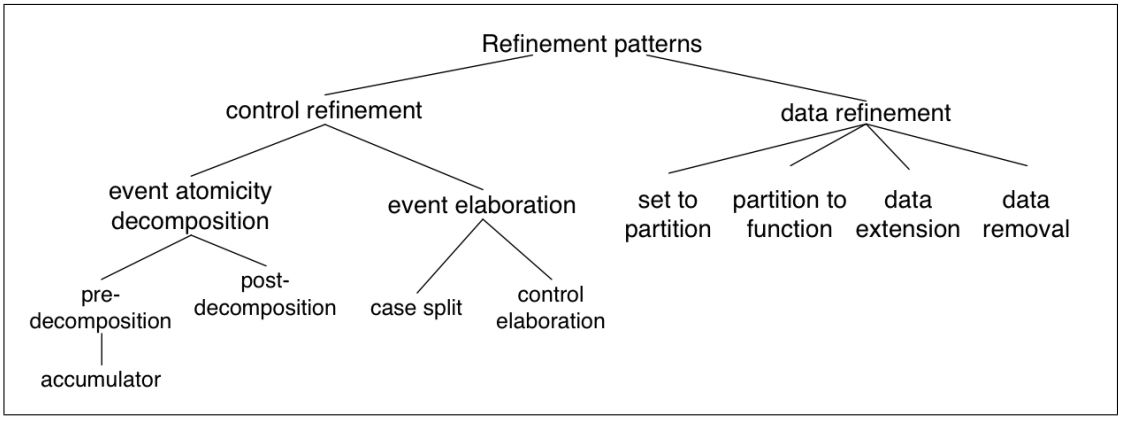


Figure 6.4: A hierarchical classification of common refinement patterns.

1. ***Control refinement:*** deals with steps that introduce details of the functionalities of the system. These details usually refer to constraints or conditions that must be imposed through a system process, or the introduction of intermediate operations required by a protocol, etc. These are known as *horizontal refinements* [3].

   (a) ***Event atomicity decomposition:*** deals with steps in which the atomicity of an abstract event is broken in the concrete model through a sequence of new events. Breaking the atomicity of an event implies the need of pre- or post-processing the data associated with an event.

      i. ***Pre-decomposition:*** handles the case when the data associated with an abstract atomic event requires to be preprocessed at the concrete level before

the event is executed. In other words, the data must reach a required state for the event to be enabled. For instance, file system models in which the data must be stored in a temporary buffer before being stored in the main unit.

- ***Accumulator:*** deals with steps in which actions of an abstract atomic event are performed in the concrete model via iteration by gradually accumulating the abstract assigned values.

ii. ***Post-decomposition:*** makes reference to steps in which the refinement of an abstract event results in the need of new events required to process the effects of the refinement.

(b) ***Event elaboration:*** relates to steps in which the refinement of an abstract event does not imply breaking its atomicity. That is, when new conditions or effects are added to the application of an existing event by means of extensions to the state.

i. ***Case split:*** handle the cases when the abstract event is refined in the concrete model by two or more events. These refinements result from a change in the state that implies distinct conditions and/or effects in the execution of an existing operation.

ii. ***Control elaboration:*** handles the cases when the abstract event is refined by only one concrete event.

2. ***Data refinement:*** deals with steps in which the state of the system is transformed; in other words, when data from the abstract level is replaced by new data at the concrete level. Data refinement usually takes place when a refinement involves changing the state of the system into a representation that is closer to the final implementation. These are known as *vertical refinements* [3].

(a) ***Set to partition:*** refers to steps in which an abstract variable is refined by partitioning it through a set of new variables in the concrete model.

(b) ***Partition to function:*** involves steps in which an abstract partition of variables is refined into a function in the concrete model.

(c) ***Data extension:*** refers to steps in which an abstract variable is refined into a concrete variable that extends the abstract data type in order to compile information from the system into a single variable.

(d) ***Data removal:*** involves the elimination of data from the abstract level that is not used to control the operation of any event at the concrete level.

Note that as progress is made down the hierarchy, the more specific the patterns are. Refinement plans are usually developed for the most specific patterns: the leaf nodes in

the hierarchy. However, when analysing a design, the generality at the top levels of this classification provides a better understanding of the user intentions in a refinement step when this one does not match any of the specific patterns. This mechanism also opens up the possibility of learning new refinement patterns; that is, when a match is not found with one of the patterns in the leaves, the details of the refinement step could be logged for the search of new patterns–however, this is not in the scope of this work. The classification of patterns as a hierarchy also facilitates the automatic matching process. We come back to this in Chapter 7.

6.1.3 Refinement plans mechanism

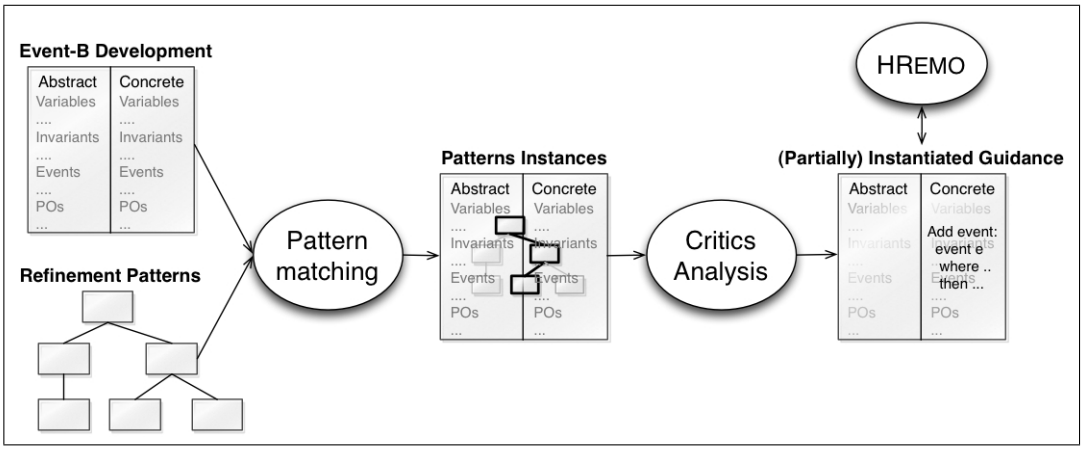


Figure 6.5: Refinement plans approach.

Figure 6.5 illustrates the process undertaken by refinement plans. Given a development, refinement plans classify refinement steps by finding matches between the step and the patterns in the hierarchy. The patterns are described in terms of abstract and concrete models, as well as the associated patterns of POs. A refinement plan is formally specified through a declarative representation as shown in Figure 6.6 – Appendix A.2 contains this formalisation for all the refinement patterns identified in the hierarchy presented above.

Each plan has a set of *arguments* which represent the main elements of the pattern associated with the plan. They specify the minimum set of elements that should be present in a refinement step in order to get a (partial) match of the pattern. Other elements associated with a pattern, such as invariants, event guards, etc., are identified only if there is a failure associated with the pattern. Moreover, the plan receives two inputs: the *Models*, which is the set of abstract and concrete models involved in a refinement step, and an *Instance* that represents the instance of the parent pattern (if one exists according to the hierarchy). Each plan also contains a set of *Preconditions* used to determine if the input models contain an

Plan_Name(*arguments*)
Inputs:
Models: set of models involved in the refinement step being analysed.
Instance: instance of the parent refinement pattern.
Preconditions: conditions that determine if the input models contain an instance of the pattern.
POs: set of PO patterns associated with the refinement pattern.
Critics: set of common patterns of errors associated with the refinement pattern.

Figure 6.6: Refinement plans declarative representation.

(partial) instance of the refinement pattern. These preconditions describe the features of the set of arguments associated with the plan. Furthermore, the *POs* component capture the patterns of proof obligations that are related to the refinement plan, while partial matches and common patterns of failure are managed by the *Critics* component.

Critics are particularly useful in situations where a development breaks down. In such situations the plans attempt to automatically generate modelling guidance to overcome the failure. This is achieved by:

1. identifying which of the known patterns are closely aligned to the given flawed refinement step, and

2. identifying if a common pattern of failure matches the failure associated with the refinement step.

This *critics* style exception mechanism is analogous to proof critics [68], a technique that analyses failed proof attempts in order to guide the search for a correct proof. In the context of refinement plans, a critic represents a common pattern of failure at the level of POs and refinements. Specifically, a critic is represented as shown below:

Critic_Name([*models_set*, *pos_set*, *instances_set*], [*preconditions*], [*guidance*])

Observe that there are three main components associated with a critic: a list of inputs, a list of preconditions and the guidance associated with the failure. The inputs have access to the abstract and concrete models involved in a refinement step, i.e. the *models_set*, the PO failures arising from the Rodin provers for the refinement step, i.e. the *pos_set*, and a set of plan instances which are partial successes in terms of the application of the refinement patterns, i.e. the *instances_set*. Moreover, associated with each critic is a set of *preconditions* that determine the requirements for the critic to be applicable as well as generic modelling *guidance* that specifies how to overcome the failure, e.g. speculating a missing invariant, splitting an existing event, etc. The guidance is provided as a list of small modifications that

must be carried out over the model in order to correct the pattern instance. Concrete instances of the refinement plan and critics are given in the following section for the accumulator refinement plan.

Another aspect of refinement plans is their integration with HREMO. When a common pattern of failure is instantiated by a particular refinement step, the associated guidance may only be partially instantiated. To fully instantiate the guidance for a given flawed refinement requires, in general, additional search and reasoning; here is where HREMO is exploited. Whenever possible, the information from the partial instantiation is used to tailor the search for invariants and guards in HREMO in an attempt to fully instantiate the guidance – further details about this are given in Section 6.3. Note that while the failure analysis and guidance generation is automatic, the decision as to whether or not the guidance is actually applied is left to the user.

The plans for four of the leaf refinement patterns have been explored: namely, the accumulator, case split, set to partition, and partition to function patterns. Here, the details for the accumulator plan are presented – the details of the other three plans are available in Appendices B, C and D respectively.

6.2 Accumulator refinement plan

The list of meta-terms shown in Figure 6.7 are used to formalise the accumulator plan.

The declarative representation associated with the accumulator plan is given in Figure 6.8. Observe that this plan contains six arguments: a set of new variables (*newVars*), an abstract variable (*absVar*), an accumulator variable (*accVar*), an abstract event (*absEvt*), its respective refined event (*refEvt*), and an accumulator event (*accEvt*). The arguments marked with '?' are instantiated through the evaluation of the preconditions while the other arguments are obtained via the parent instance; i.e. they are instantiated by the preconditions of the *event pre-decomposition* plan – which is the parent plan as specified by the inputs of the accumulator plan. Moreover, two preconditions are specified in order to identify an (partial) instance of the pattern:

- P1 specifies that an accumulator event *accEvt* should be found among the *predecessors* of the refined event; i.e. an event that modifies the accumulator variable *accVar* through the action pattern $accVar := accVar \oplus \alpha$ should be part of the set of events that enable the refined event *refEvt*, and

- P2 determines that the refined event *refEvt* must assign the accumulator variable *accVar* to the abstract variable *absVar*; i.e. the accumulated value should be assigned in the refinement of an action in the refined event.

action(act, evt)	$\widehat{=}$	true if *act* is an action of event *evt*
element_of(x, set)	$\widehat{=}$	true if *x* is an element of *set*
is_instance_of(i, pattern_name)	$\widehat{=}$	true if *i* is an instance of pattern *pattern_name*
is_deadlocked(evt)	$\widehat{=}$	true if event *evt* is deadlocked
is_empty_set(x)	$\widehat{=}$	true if *x* is an empty set
addEvent(event(pars, grds, actions), M)	$\widehat{=}$	an event with parameters *pars*, guards *grds* and actions *actions* is added to model *M*
grd2Sub(grd)	$\widehat{=}$	transforms predicate *grd* into a substitution
is_guard(g, evt)	$\widehat{=}$	true if *g* is a guard of event *evt*
po_type(po, T, evt)	$\widehat{=}$	true if the proof obligation *po* is of type *T* and is associated with event *evt*
provable(P)	$\widehat{=}$	true if predicate *P* is provable
addGuard(g, evt)	$\widehat{=}$	a guard *g* is added to event *evt*
is_invariant(inv, M)	$\widehat{=}$	true if *inv* is an invariant of model *M*
addInvariant(inv, M)	$\widehat{=}$	an invariant *inv* is added to model *M*
po_abstract_action(act, po)	$\widehat{=}$	true if an action *act* of the abstract model is associated with the proof obligation *po*
po_concrete_action(act, po)	$\widehat{=}$	true if an action *act* of the concrete model is associated with the proof obligation *po*
addVariable(v, T, M)	$\widehat{=}$	a variable *v* with type *T* is added to model *M*
makeRefine(refEvt, absEvt)	$\widehat{=}$	makes the event *refEvt* refine the abstract event *absEvt*
replaceAction(oldAct, newAct, evt)	$\widehat{=}$	replaces action *oldAct* with a new action *newAct* in event *evt*

Figure 6.7: Critics meta-terms.

Accumulator(newVars, ?absVar, ?accVar, absEvt, refEvt, ?accEvt)
Inputs:
Models: {AM, CM}
Instance: Event_PreDecomposition(newVars, absEvt, refEvt, predecessors)
Preconditions:
P1. ?accEvt∈predecessors ∧ ?accVar∈newVars ∧
action(?accVar := ?accVar⊕α, ?accEvt)
P2. ?absVar∈V(AM) ∧ action(?absVar := F(?accVar), refEvt)
POs: {INV_PO_Init_Event, INV_PO_Accumulator_Event,
INV_PO_Refined_Event, SIM_PO_Refined_Event}
Critics: {initialisation_speculation, loopGuard_speculation, postGuard_speculation,
invariant_speculation, accumulator_speculation}

Figure 6.8: Declarative representation of the accumulator decomposition plan.

These two preconditions identify if an atomic event is refined through an accumulation at the concrete level of a model. Note that the details of the PO patterns associated with the accumulator plan were introduced in Section 6.1; therefore, they are only referenced here by

their name. The critics are also referred by their name only; however, the following section provides details for each of them.

6.2.1 Critics

The focus of this section is on the critics aspect of the accumulator refinement plan, and how modelling guidance is automatically generated via partial matching and failure analysis. The critics mechanism handles common patterns of failures associated with the refinement patterns. It will be noted that typically the guidance is only partially instantiated; i.e. the suggested guidance will be in the form of partially instantiated schemas. When possible, HREMO should be used to fully instantiate these schemas. This integration with HREMO will be explored in Section 6.3.

Five critics associated with the accumulator refinement plan have been identified. The meta-terms presented in Figure 6.7 are used in order to specify the critics. Moreover, the guidance may refer to elements presented in the modelling schema shown previously in Figure 6.2. These elements are identified by the critic; however, the details of their identification is omitted here not to overload the presentation of the critics.

Initialisation_speculation critic:

Initialisation_speculation_critic (
[{AM, CM}, po_set, instances_set],
[P1: element_of(I, instances_set) ∧ is_instance_of(I, "*accumulator*"),
P2: ¬is_deadlocked(I.accEvt),
P3: is_empty_set(predecessors(I.accEvt))],
addEvent(event(_,[H_0], [accVar:=Init, grd2Sub(H_1)]), CM)
)

This critic triggers when the accumulation process does not have an initialisation phase or the initialisation fails. This critic is applicable iff:

P1. An accumulator pattern instance *I* is detected within the set *instances_set*.

P2. The accumulator event *accEvt* associated with instance *I* deadlocks.

P3. The accumulator event *accEvt* does not have predecessors; i.e. there are no events whose execution enable the accumulator event.

Verifying that the refinement step contains an instance of the accumulator pattern is the first condition that must hold in order to apply the critic. Precondition 2 checks if the accumulator event deadlocks; which would suggests a possible failure in the initialisation. Finally,

precondition 3 verifies that in fact the deadlock of the accumulator event occurs because of an error associated with the initialisation event – since the deadlock may occur because of other errors in the model; e.g. failure to handle the termination of the accumulation process. If these preconditions succeed, the guidance suggested to the user is the addition of the initialisation event to the concrete model; i.e. an event with shape:

$$\textbf{Event}\, C_i \,\widehat{=}\, \textbf{when}\ H_0\ \textbf{then}\ accVar := Init \parallel grd2Sub(H_1)$$

note that H_0 and H_1 refer to elements of the accumulator pattern introduced in Figure 6.2.

LoopGuard_speculation critic:

LoopGuard_speculation_critic (
[{AM, CM}, po_set, instance_set],
[P1: element_of(I, instance_set) ∧ is_instance_of(I, "*accumulator*"),
P2: ¬is_guard(G$_{\prec}$($\overline{W}$, $\overline{V_k}$), I.accEvt) ∨
(element_of(po, po_set) ∧ po_type(po, INV, I.accEvt) ∧ ¬provable(po))],
addGuard(G$_{\prec}$($\overline{W}$, $\overline{V_k}$), I.accEvt)
)

This critic triggers when there are incorrect or missing guards associated with the accumulator event. This critic is applicable iff:

P1. An accumulator pattern instance *I* is detected within the set *instances_set*.

P2. The accumulator guard of the accumulator event *accEvt* associated with instance *I* is missing; or is not compatible with the pattern *G$_{\prec}$($\overline{W}$, $\overline{V_k}$)*; or there is a failed invariant preservation proof obligation *po* in the set of proof obligations *po_set* that is associated with the accumulator event.

As in the previous case, the first precondition verifies that the refinement step contains an instance of the accumulator pattern; while the second precondition verifies if there is a failure associated with the guard that handles the accumulation process in the accumulator event. This condition can be verified by comparing the guards of the event with the expected guard pattern. If the pattern matches the general form but the guard is incorrect, a failed invariant preservation PO would be produced. The suggested guidance specifies that a guard with the shape G$_{\prec}$($\overline{W}$, $\overline{V_k}$) should be added to the accumulator event.

PostGuard_speculation critic:

PostGuard_speculation_critic (
[{AM, CM}, po_set, instance_set],
[P1: element_of(I, instance_set) ∧ is_instance_of(I, "*accumulator*"),
P2: element_of(po, po_set) ∧ po_type(po, SIM, I.refEvt) ∧ ¬provable(po),
P3: ¬is_guard($G_{=}(\overline{W}, \overline{V_k})$, I.refEvt)],
addGuard($G_{=}(\overline{W}, \overline{V_k})$, I.refEvt)
)

This critic triggers when the guard of the refined event, which ensures the accumulation process is complete, is either incorrect or missing. This critic is applicable iff:

P1. An accumulator pattern instance *I* is detected within the set *instances_set*.

P2. There is a failed simulation proof obligation *po* in the set of proof obligations *po_set*, which is associated with the refined event *refEvt* of instance *I*.

P3. The post-accumulator guard of the refined event *refEvt* is missing or it is not compatible with the guard pattern $G_{=}(\overline{W}, \overline{V_k})$.

The first precondition verifies that an instance of the accumulator pattern is identified in the refinement step. Then, if the guard of the refined event is wrong or missing, a failed simulation PO would be generated. This is verified through the second precondition. However, the failure of the simulation PO can be associated with different causes; for instance, the absence of the gluing invariant. The third precondition examines if the failure is associated with the post-accumulator guard by comparing the guards of the refined event with the expected guard pattern. If the three preconditions succeed the suggested guidance is the introduction of a guard with the shape $G_{=}(\overline{W}, \overline{V_k})$ to the refined event.

Invariant_speculation critic:

Invariant_speculation_critic (
[{AM, CM}, po_set, instance_set],
[P1: element_of(I, instance_set) ∧ is_instance_of(I, "*accumulator*"),
P2: element_of(po, po_set) ∧ po_type(po, SIM, I.refEvt) ∧ ¬provable(po),
P3: is_guard($G_{=}(\overline{W}, \overline{V_k})$, I.refEvt),
P4: ¬is_invariant($H_1 \Rightarrow F_{\leq}(\overline{W}, \overline{V_k})$, CM)],
addInvariant($H_1 \Rightarrow F_{\leq}(\overline{W}, \overline{V_k})$, CM)
)

This critic handles the case when the accumulator invariant is incorrect or missing. This critic is applicable iff:

P1. An accumulator pattern instance *I* is detected within the set *instances_set*.

P2. There is a failed simulation proof obligation *po* in the set of proof obligations *po_set*, which is associated with the refined event *refEvt* of instance *I*.

P3. The post-accumulator guard of the refined event *refEvt* is not missing and it is compatible with the guard pattern $G_{=}(\overline{W}, \overline{V_k})$.

P4. The accumulator invariant is missing or it is not compatible with the invariant pattern $H_1 \Rightarrow F_{\leq}(\overline{W}, \overline{V_k})$.

Precondition 1 verifies that an instance of the accumulator pattern is identified in the refinement step. As with the *postGuard_speculation critic*, if the gluing invariant is wrong or missing, a failed simulation PO would be generated. The presence of the PO is verified through precondition 2, while precondition 3 discards the failure being associated with the post-accumulator guard. Finally, precondition 4 examines if the failure is associated with the invariant by comparing the invariants of the concrete model with the expected invariant pattern. If all the preconditions succeed the suggested guidance is the introduction of an invariant with shape $H_1 \Rightarrow F_{\leq}(\overline{W}, \overline{V_k})$ to the concrete model.

Accumulator_speculation critic:

Accumulator_speculation_critic (
[{AM, CM}, po_set, instance_set],
[P1: element_of(I, instance_set) ∧ is_instance_of(I, "*control_elaboration*"),
P2: element_of(po, po_set) ∧ po_type(po, SIM, I.refEvt) ∧ ¬provable(po),
P3: po_abstract_action(V:=β, po) ∧ po_concrete_action(V:=V⊕α, po)],
[addVariable(accVar, β.Type, CM),
addVariable(start, boolean, CM),
makeRefine(I.refEvt, *skip*),
replaceAction(V:=V⊕α,accVar:=accVar⊕α, I.refEvt),
addGuard(start=TRUE, I.refEvt),
addEvent(event(_, [start=TRUE], [V:=f(accVar), start:=FALSE]), CM),
addEvent(event(_, [start=FALSE], [accVar:=Init, start:=TRUE]), CM)])
)

This critic handles the case when an accumulator event refines an abstract event whose actions are performed in an atomic step.

P1. A control elaboration pattern instance *I* is detected within the set *instances_set*.

P2. There is a failed simulation proof obligation *po* in the set of proof obligations *po_set*, which is associated with the refined event *refEvt* of instance *I*.

P3. The abstract and refined actions associated with the failure have the form V:=β and V:=V$\oplus\alpha$, respectively.

The first precondition verifies that an instance of the control elaboration pattern is identified in the refinement step. The second precondition checks if a failed simulation PO has been generated. Finally, the third precondition verifies if the failure is associated with the refinement of an atomic action into an accumulation action. If an abstract atomic event is refined through accumulation a failure would arise. That is because it is not possible to prove that executing the abstract event results in the same outcome than executing the event at the concrete level. If all the preconditions succeed the suggested guidance is to transform the control elaboration instance into an accumulator pattern. In order to achieve this, a set of steps must be carried out. That is:

- Add an accumulator variable to the concrete model whose type is determined by the assignment in the abstract action.
- Add a new boolean variable to control the start and end of the accumulation process.
- Transform the refined event into an accumulator event, i.e. the event should refine skip, the assignment V:=V$\oplus\alpha$ should be replaced by accVar:=accVar$\oplus\alpha$ and a guard should be added that specifies the accumulation process has started.
- Create a new refined event with a guard verifying the accumulation process started and with actions that assigned the accumulated value and specify the end of the process.
- Create an initialisation event with a guard verifying the accumulation process has not started and with actions that initialise the accumulator variable and specifies the beginning of the process – note that the initialisation value must be supplied by the user.

6.2.2 Example application of the accumulator plan

In this section the application of the critics mechanism is illustrated by analysing a flawed version of the addition model shown in Figure 6.1. The version of the model that contains the error is shown in Figure 6.9. The abstract model adds the value of y to x in an atomic step, while at the concrete level the assignment is intended to be performed gradually in the refined event. As can be observe, the model produces a failed SIM PO.

Following the classification hierarchy, the model is classified as an instance of the *control_elaboration* refinement pattern–since the refinement step refines an existing event via a new variable and the refinement does not involve new events or case splits–see Appendix A.2 for details about the preconditions of the control elaboration pattern. As it stands, the *accumulator_speculation critic* is triggered as follows:

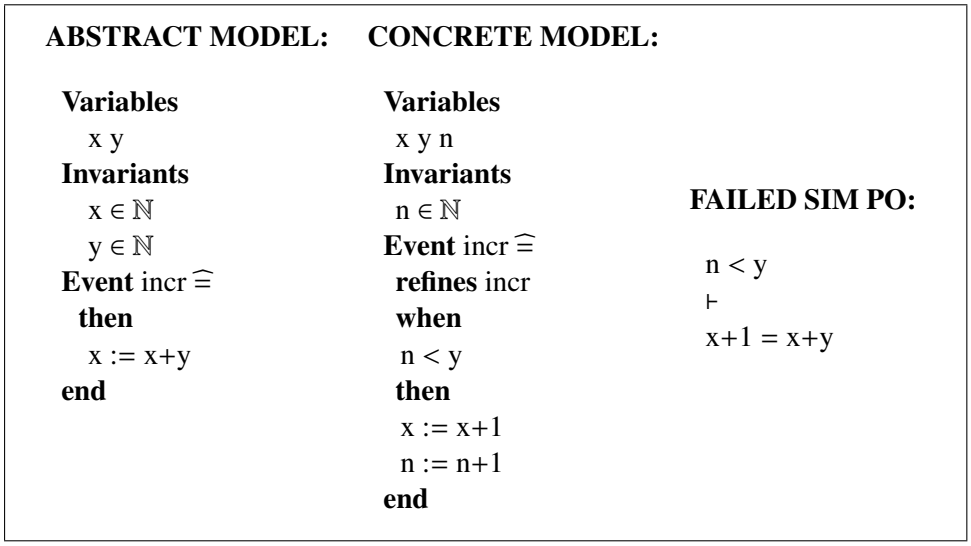


Figure 6.9: Flawed accumulator refinement pattern instance.

Accumulator_speculation critic: the conditions yield:

P1. The model is an instance of the control elaboration pattern.

P2. A failed simulation PO associated with the refined event occurs, i.e.:

$$n < y \vdash x + 1 = x + y.$$

P3. The abstract and refined actions associated with the failure have the form V:=β and V:=V$\oplus\alpha$, respectively. With respect to the example that is:

Abstract action: $x := x + y$

Concrete action: $x := x + 1$

Guidance

An instance of the accumulator refinement pattern is speculated and suggested to the user. The suggested model is shown in Figure 6.10. The modifications applied to the model involved:

1. Adding an accumulator variable x_2 to the refined model.
2. Adding a new boolean variable *start*, conditions *start=FALSE* and *start=TRUE*, and actions *start:=TRUE* and *start:=FALSE* to specify the start and termination of the accumulation process.
3. Transforming the existing event into an accumulator event, i.e. the event refines skip and the assignment x := x + 1 is replaced by $x_2 := x_2 + 1$.
4. Creating a new refined event with action $x := x_2$.

5. Adding an initialisation event with action x_2 := *InitValue* – initialisation value to be supplied by the user.

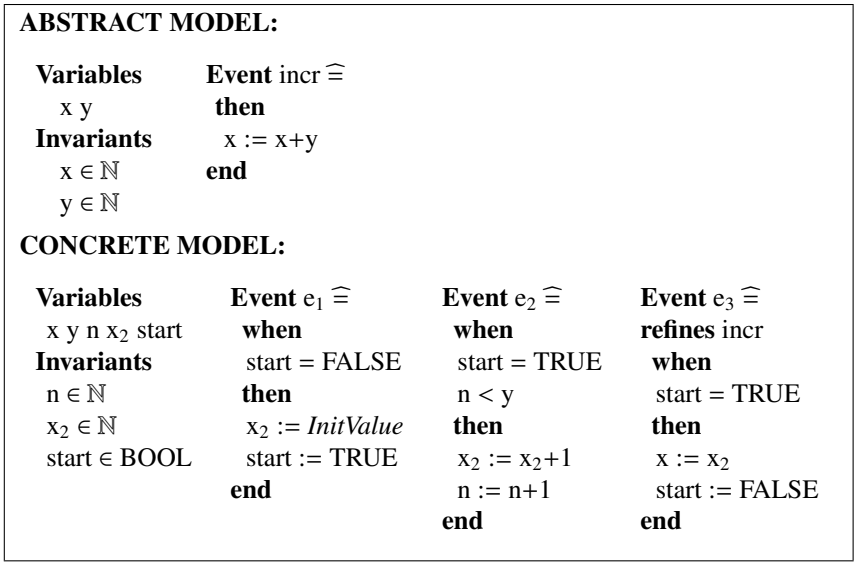


Figure 6.10: Guidance.

The guidance currently provided is in the form of partial instantiations of the pattern schema; thus, user interaction is required. Let us assume the user provides the instantiation of the pattern shown in Figure 6.11 – where the user has added the initialisation of variable *n* and the initial value for the temporal variable x_2.

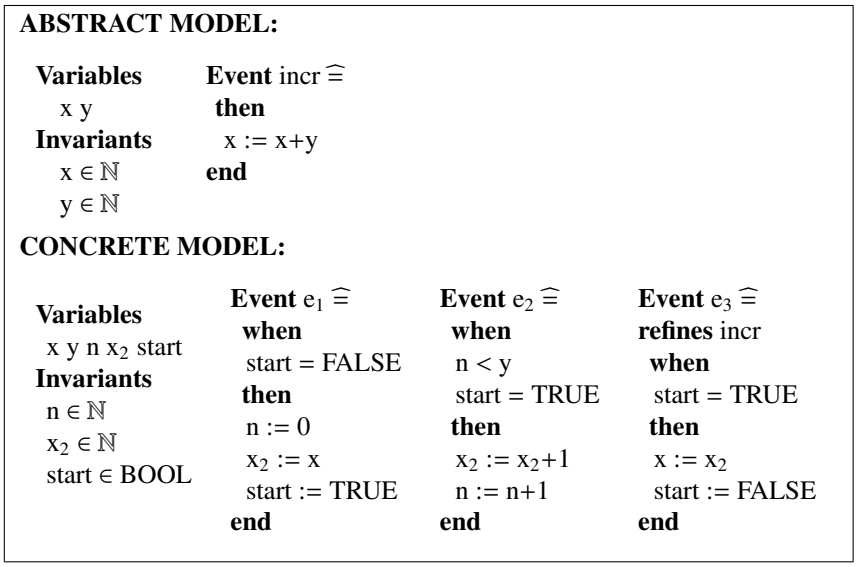


Figure 6.11: User provided partial instantiation.

At this point the *postGuard_speculation* critic triggers with the following instantiation:

PostGuard_speculation critic: the conditions yield:

P1. The accumulator pattern is identified.

P2. A simulation PO pattern associated with the refined event fails, i.e.

$$start = TRUE \vdash x_2 = x + y.$$

P3. The post-accumulator guard is missing.

Guidance:
Add a guard to event *end_ok* with the form:

$$G_{=}(x_2, n, start, x, y).$$

We will revisit this guard schema below, and describe how it can be automatically instantiated. For now assume that the correct instantiation is available, i.e. $n = y$. Because the invariant is also missing, the failure persists, this triggers the *invariant_speculation* critic.

Invariant_speculation critic: the conditions yield:

P1. An accumulator pattern is identified.

P2. A simulation PO pattern associated with the refined event fails, i.e.

$$start = TRUE, n = y \vdash x_2 = x + y.$$

P3. The post-accumulator guard $n = y$ is present in the refined event and is compatible with the guard pattern.

P4. The accumulator invariant is missing from the model.

Guidance:
Add an invariant to the model with the shape:

$$(start = TRUE) \Rightarrow F_{\leq}(x_2, n, x, y).$$

Note that due to use of natural numbers, $\leq$ is instantiated to $\leq$.

Note also that the guidance currently provided is again in the form of partial instantiations of the schemas. At this point, there are three options to find the correct instantiation: i) through interaction with the user, ii) through the use of proof patterns, or iii) through the use of automated theory formation (ATF). Option three is further explored in the following section.

6.3 Combining Refinement Plans and HRemo

In the previous section the application of the accumulator pattern for the addition example yielded partial instantiations of missing invariants and guards. Using HREMO on its own fails to find the missing invariant after the default 1000 theory formation steps. This does not imply that the invariant cannot be found, rather it means that additional search is required. By combining refinement plans and event error traces with HREMO these issues can be effectively addressed. In this section we report on an experiment in which HREMO is used to fully instantiate the invariant and guard templates obtained in the example of the previous section.

The process of finding a "correct" refinement typically involves exploring many incorrect models. Refinement plans aim at providing guidance when a failed refinement is closely aligned with a known pattern. However, refinement plans are limited by the observed patterns. On the other hand, as mentioned above, HREMO also exhibits some limitations. In order to overcome these limitations we combine both approaches, in particular, we extend the invariant discovery approach by:

- using the ProB animator [86] to generate traces that contain undesirable states which can be used by HREMO to find missing guards, and
- using the patterns of invariants and guards available in the refinement plans to automatically tailor the search in HREMO.

6.3.1 Illustrative example

Recall the Event-B model shown in Section 3.1.1. We use this example to illustrate how HREMO can identify a missing guard. Figure 6.12 shows a modified fragment of the original model which specifies the refinement of event *addA* – at the abstract level variable *full* is modified through a non-deterministic substitution *full* :∈ *BOOL* when *full* is false, while the concrete event gradually increments variable *x* by one unit.

Abstract event:	**Concrete Event:**
Event addA $\widehat{=}$	**Event** addC $\widehat{=}$
when	**refines** addA
full = false	**then**
then	x := x+1
full :∈ BOOL	**end**
end	

Figure 6.12: Flawed refined event.

As it stands the model generates the failed guard strengthening PO shown below:

$$\top \vdash full = false$$

Note that the PO fails because the guard of the concrete event has not been defined.

The model is identified as an instance of the *control elaboration* refinement plan since the abstract event is not split at the concrete level and the refined event remains atomic (see Appendix A.2 for the details of this plan). A critic associated with the refinement plan is triggered and the provided guidance is the addition of a guard to the refined event – this is identified by the critic through the inspection of the failed PO and the simulation trace, which specifies an inconsistency between the guards of the abstract and concrete event as it will be explained next.

The animation trace shown in Figure 6.13 is produced by the ProB simulator for the modified version of the model. Note that the trace contains an incorrect step; i.e. *S4*. This is specified by ProB which detects that the abstract guard *full=false* is not satisfiable by the guard of the refined event once step *S4* has been carried out. Based on this simulation trace and the modified Event-B model, the core concepts are extracted as shown in Figure 6.14. Note that an extra concept is added to the input background information given to HR. That is, the concept *good*, which specifies the states of the simulation trace that do not contain errors. The purpose of this is to look for the missing guard(s) within the conjectures associated with concept *good* – since the execution of the event should only produce correct steps.

		Animation steps			
		Correct steps			**Incorrect step**
	Variables	**S1**	**S2**	**S3**	**S4**
Abstract	full	false	false	false	true
Concrete	x	0	1	2	3
	m	3	3	3	3

Figure 6.13: Animation trace generated by the ProB simulator.

state(A)
S1
S2
S3
S4

good(A)
S1
S2
S3

boolean(B)
true
false

integer(C)
0
1
2
3
4

full(A,B)	
S1	false
S2	false
S3	false
S4	true

x(A,C)	
S1	0
S2	1
S3	2
S4	3

m(A,C)	
S1	3
S2	3
S3	3
S4	3

Figure 6.14: Core concepts.

After running HR for 1000 theory formation steps a total of 41 conjectures are generated. Through manual inspection, the equivalence conjectures associated with concept *good* are

examined in order to find the missing guard. Only equivalence conjectures are explored since the missing guard only occurs within the correct steps. The following conjectures are identified:

$$\forall A.state(A) \wedge good(A) \Leftrightarrow boolean(FALSE) \wedge full(A,FALSE)$$
$$\forall A,B,C.state(A) \wedge good(A) \Leftrightarrow integer(B) \wedge m(A,B) \wedge integer(C) \wedge x(A,C) \wedge B>C$$
$$\forall A,B,C.state(A) \wedge good(A) \Leftrightarrow integer(B) \wedge x(A,B) \wedge integer(C) \wedge m(A,C) \wedge B<C$$
$$\forall A.state(A) \wedge good(A) \Leftrightarrow \neg(boolean(TRUE) \wedge full(A,TRUE))$$

The right hand side of the conjectures represent the potential guards. These are shown below in their corresponding Event-B representation. Note that (6.1) and (6.4) cannot be selected as guards since *full* is an abstract variable that disappears at the concrete level. On the other hand, (6.2) and (6.3) are valid predicates and both represent the missing guard – since they are equivalent predicates.

$$full=false \tag{6.1}$$
$$m>x \tag{6.2}$$
$$x<m \tag{6.3}$$
$$\neg(full=true) \tag{6.4}$$

This illustrates the integration between HREMO and refinement plans. That is, through the use of refinement plans it was identified that a guard was missing in the model, while HREMO was used to automatically generate the missing guard.

In the following section new heuristics are described which tailor HREMO according to the guidance generated by the refinement plans in order to discover invariants and guards.

6.3.2 Extending the heuristic approach

As described in Section 3.3, two type of heuristics are used by HREMO, configuration heuristics (CH) and selection heuristics (SH). When a pattern of an invariant or a guard is available then the following heuristics are applied:

Configuration heuristics:

CH1. *Prioritise core and non-core concepts expected in the pattern of the invariant or guard.*

CH2. *Follow with core and non-core concepts that occur within failed POs.*

CH3. *Generate conjectures that are compatible with the type of the expected invariant. If*

looking for a guard, generate only equivalence conjectures unless there is a failed GRD PO associated with the event, in which case, implication conjectures are also formed. Moreover, if possible, prioritise the concept associated with the abstract guard (as a core or non-core concept as it fits).

CH4. *Select only production rules which will give rise to conjectures relating to the type of the expected invariant or guard.*

Equivalence conjectures are always generated since this optimises the theory formation process [32].

The selection heuristics for the search of invariants based on patterns are the same as those applied previously in Chapter 3. This requires selecting conjectures where the sets of variables occurring on the left- and right-hand sides are disjoint, selecting the most general conjectures, and selecting the conjectures that discharge the failed POs and that minimise the number of additional proof failures. Note that here the selection of conjectures is focused in the core and non-core concepts that relate to the invariant pattern, as opposed to the heuristics presented in Chapter 3, which focused on core and non-core concepts from the failed POs.

In the case of missing guards the selection process differs. Through the use of the ProB animator it is possible to detect event errors which result in traces that contain undesirable states. When a trace of this type is generated we provide HREMO with the concept of *good* states, as it was shown above, which are the steps of the trace with no associated event errors. The selection is then focused on conjectures that express equivalences with the concept of *good*, i.e. conjectures of the form:

$$good \Leftrightarrow \phi$$

where ϕ represents the potential missing guard. Moreover, if there is a failed GRD PO associated with the event, and the abstract guard could be represented as a core or non-core concept, HREMO searches for conjectures of the form:

$$\phi \Rightarrow \alpha$$

where α represents the abstract guard.

Recall the application of the *postGuard_speculation* and *invariant_speculation* critics presented in Section 6.2.2 resulted in partially instantiated guard and invariant schemas. These schemas are used to tailor HREMO in the search for the missing guard and invariant. The instantiated guard schema obtained by the *postGuard_speculation* critic was:

$$G_{=}(x_2, n, start, x, y)$$

Based on this, the configuration heuristics are instantiated as follows:

CH1: Prioritised concepts from the guard schema: x_2, n, *start*, x and y.

CH2: Concepts from the failed POs: $x+y$, $x_2=x+y$ and *start=TRUE*.

CH3: Searching for a guard and there is no failed GRD PO associated with the event; thus, only equivalence conjectures are generated.

CH4: As the variables involved in the guard are natural numbers, the production rules to be used are the *arithmetic* and *numrelation* PRs (no PR is associated with *start* since it is a boolean variable).

After 65 seconds and 1000 theory formation steps HREMO returns 1 conjecture:

$$good \Leftrightarrow y = n$$

which means that the missing guard is y = n. A similar approach is followed in the search for the missing invariant. Recall the invariant template instantiated by the *invariant_speculation* critic of the accumulator plan:

$$(start = TRUE) \Rightarrow F_{\leq}(x_2, n, x, y).$$

Based on this, the configuration heuristics are instantiated as follows:

CH1: Prioritised concepts from the invariant schema: *start*, x_2, n, x and y.

CH2: Concepts from the failed POs: *start=TRUE*, $x+y$, and $x_2=x+y$.

CH3: The invariant template suggests the search for an implication. Therefore, implication conjectures are enabled during the search.

CH4: As the variables involved in the invariant template are natural numbers, the production rules to be used are the *arithmetic* and *numrelation* PRs (no PR is associated with *start* since it is a boolean variable).

After 45 seconds and 1000 theory formation steps HREMO returns 1 conjecture:

$$start = TRUE \Rightarrow x_2 = x + n$$

which represents the missing invariant.

6.4 Summary

This chapter presented Refinement Plans, an approach to the classification of refinement steps as well as for the generation of guidance when a refinement step is flawed but is close to a known pattern of refinement. Each refinement plan is composed of a *modelling pattern*, a set of *PO patterns*, and a set of *critics* – where a critic represents a common pattern of

failure at the level of refinements and POs. Associated with each critic is generic modelling guidance as how to overcome the failure. When a common pattern of failure is instantiated by a particular refinement step, the associated guidance is provided to the user. In cases in which the guidance is only partially instantiated, we have experimented with HREMO as a mechanism to fully instantiate it. This is achieved by using the information of the partial instantiation to tailor the search within HREMO.

Chapter 7

Refinement plans: workbench and results

In this chapter REMO is presented. A prototype system that implements the ideas behind *reasoned modelling*. In particular, it implements refinement plans as well as reasoned modelling critics [70], a technique that extends the notion of proof critics [68] by exploiting the way in which proof failure analysis typically informs the activity of modelling. In the following section the architecture of REMO is presented as well as the results obtained from a series of experiments carried out over a set of Event-B developments.

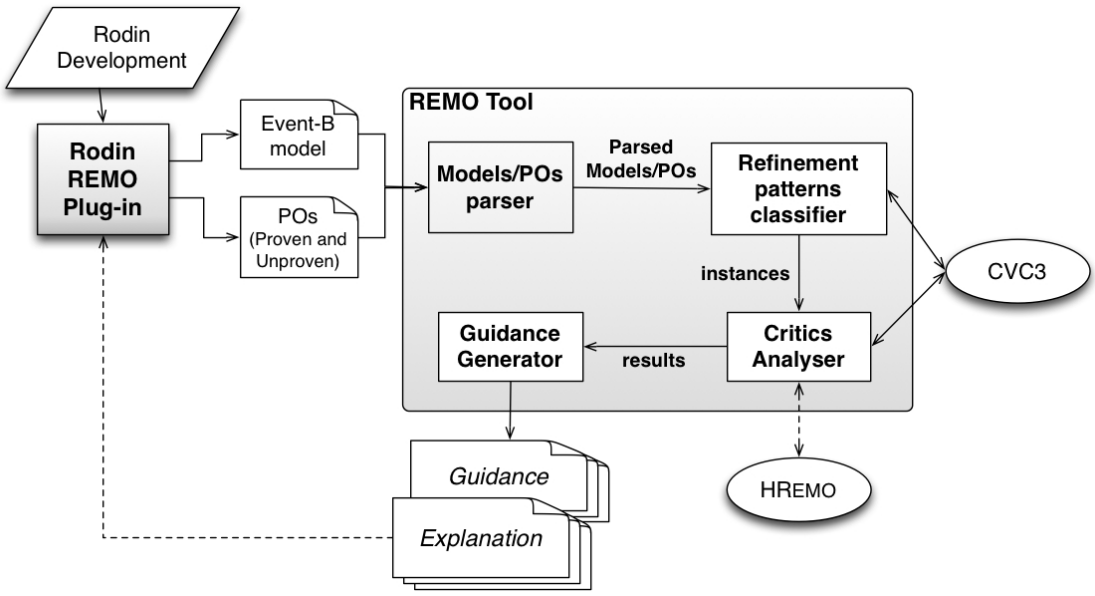
7.1 Tool Architecture

An architectural view of the REMO tool is shown in Figure 7.1. The *Rodin REMO plug-in* provides the interface between the Rodin toolset and the REMO tool. A development is parsed by the *Models/POs parser* and is then passed to the *Refinement plans classifier* that classifies the patterns of refinement used in a development. The role of the *Critics analyser* is to find ways of overcoming failures via the critics mechanism. The classifier and the analyser interact with CVC3 when a precondition requires proof. The analyser also interacts with HREMO in order to search for missing/wrong invariants or guards. The raw results are passed to the *Guidance generator*, which produces a list of alternative guidance suggestions. The guidance is then sent to the *Rodin Remo plug-in* and presented to the user – note that the stippled lines indicate work in progress. Detailed descriptions of each component are provided in the following sections.

Since the ideas of reasoned modelling have been proposed for Event-B, REMO is necessarily integrated with the Eclipse-based Rodin toolset — implemented in Java. However, the core of REMO has been implemented in OCaml (*Objective Caml Language*)[1]. This has been motivated by the following factors:

[1] See `http://caml.inria.fr/ocaml/`

- this is a prototype implementation which is expected will change (frequently). OCaml is a declarative functional programming language which reduces the size of the source code, thus making change to the code (relatively) easy;
- a large part of the work is checking preconditions, which can be handled very elegantly using inductive data types and pattern matching, which are supported by OCaml;
- compared to other languages in the ML-family, OCaml embodies an object-oriented layer, which provides a good encapsulation mechanism to represent models. In particular, the object-oriented feature of the language has been very useful for representing the hierarchal nature of refinement plans through the use of inheritance.



The labels with **bold** font indicate the main components of the tool chain; the stippled lines indicate work in progress; the rectangle box represents input data used in a process; the square boxes represent processes (where the white filled processes are part of the tool extension discussed here); the document boxes represent files that are used as inputs and/or outputs of a process; and the ovals represent external processes .

Figure 7.1: The REMO *tool architecture.*

7.1.1 Rodin REMO Plug-in

The goal of the Rodin REMO plug-in is to provide an interface between the Rodin toolset and the REMO tool. Specifically the plug-in has two main roles:

1. translate Event-B developments (models, contexts and proof obligations) from Rodin into the REMO tool; and

2. communicate modelling guidance and explanation back from the REMO tool into Rodin.

At the current state only role 1 has been implemented – role 2 is planned future work, and communication is currently at the level of automatically generated files. However, a description is given throughout this section of how it is envisaged the guidance will be presented to the user by the Rodin plug-in.

The input of the Rodin REMO plug-in is a Rodin development, from which the plug-in generates two files that serve as the input of the REMO tool. The first file contains a string representation of the Event-B models (machines and contexts) of a development, while the second file stores a string representation of all the POs (status, type, hypotheses and goal) associated with each model. These files are then translated into the REMO internal representation, i.e. OCaml objects.

7.1.2 The REMO Tool

As shown in Figure 7.1, REMO contains the following components: *Models/POs Parser*, *Refinement Patterns Classifier*, *Critics Analyser* and *Guidance Generator*, which are applied sequentially and discussed separately below.

7.1.2.1 Models/POs parser

The role of this component is to parse the model and PO files generated by the Rodin REMO plug-in. This is achieved through the definition of types that represent the main components of an Event-B model. That is, machine elements, event elements, predicates, expressions, etc. The parsed representation of the developments is then used by REMO in order to carry out the classification of patterns and analysis of failures associated with instances of the patterns.

7.1.2.2 Refinement patterns classifier

The classifier evaluates the parsed Event-B models in order to classify the patterns of refinement that occur in a development. Moreover, the classifier invokes the critics analyser when a failure related to a step with a pattern instance is detected.

The pattern hierarchy presented in the previous chapter (Figure 6.4) plays an important role for the identification of patterns. As mentioned before, a pattern may appear in a refinement step only if an instance of the pattern immediately above in the hierarchy has been identified. This is implemented within REMO through the use of inheritance as shown in Figure 7.2.

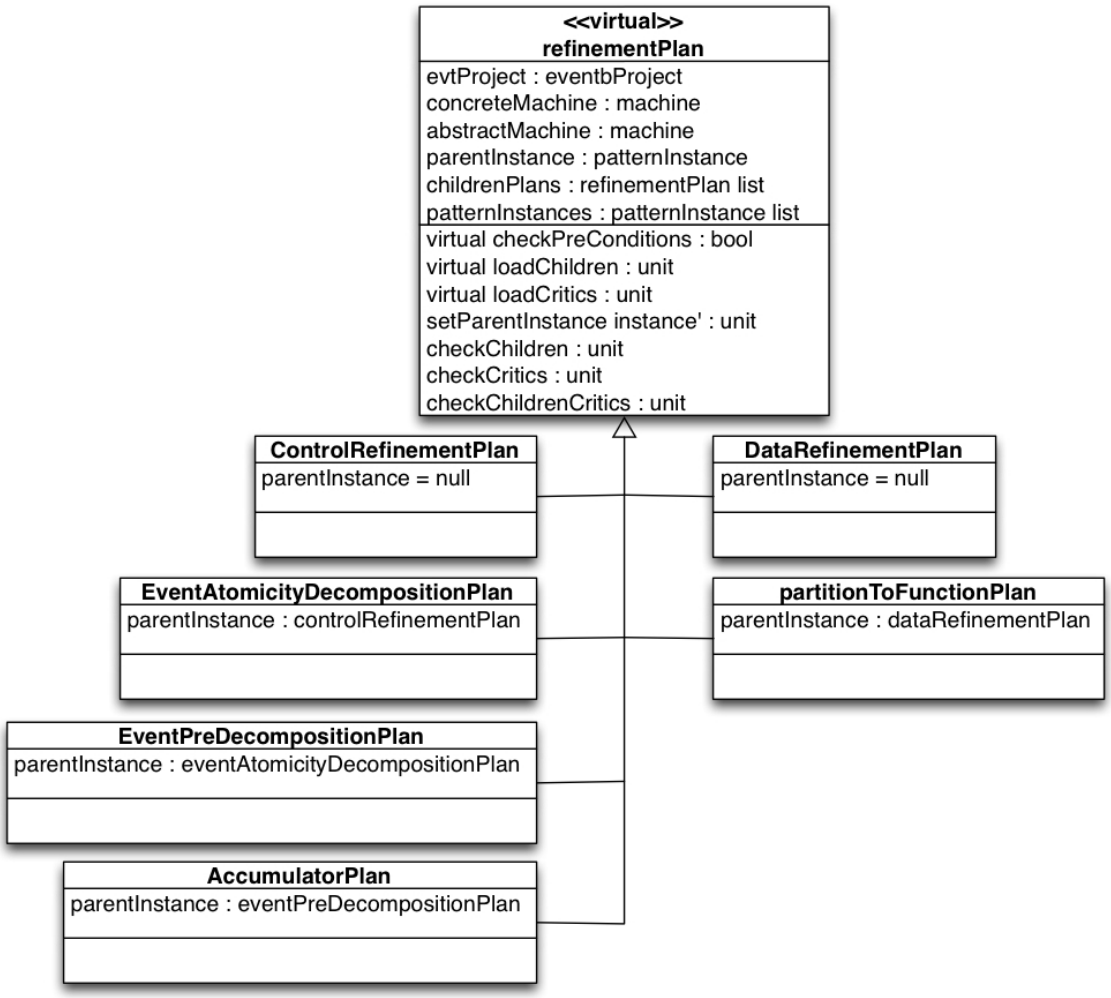


Figure 7.2: Refinement patterns classifier structure.

All refinement plans inherit from a *virtual* class called *refinementPlan*. In OCaml virtual classes have the same role as abstract classes in common OO languages; i.e. they are used in order to create subclasses that share the structure imposed by the virtual class. Each refinement plan contains a variable *parentInstance* which has the reference to the pattern instance above in the hierarchy. Moreover, each refinement plan contains a collection *childrenPlans* that defines its children plans and a collection *patternInstances* that contains the complete or partial instances of the refinement pattern found in the refinement step.

The virtual methods *checkPreconditions, loadChildren* and *loadCritics* must be implemented by each concrete instance of the *refinementPlan* class. In the method *checkPreconditions* the specific preconditions that determine if the input models contain an instance of the pattern associated with the plan must be implemented. The methods *loadChildren* and *loadCritics* must contain references to each of the children plans and the critics associated with the plan.

Algorithm 10 shows the pseudocode for the identification of the pattern associated with the accumulator refinement plan in a refinement step. This illustrates the general mechanism used to identify refinement patterns within a model.

Algorithm 10 Pseudo-code for the identification of the accumulator refinement plan.

```
1  class ACCUMULATORREFINEMENTPLAN inherits REFINEMENTPLAN
2      parentInstance ← EVENTPREDECOMPOSITIONPLAN instance
3      method checkPreconditions
4          accumEvents ← find accumulator events in parentInstance.predecessors
5          if there is an accumulator even in accumEvents then
6              for all evt in accumEvents do
7                  if atomic action from parentInstance.absEvt is refined in parentInstance.refEvt
   through accumulator variable from evt then
8                      instance ← create accumulator refinement instance
9                      add instance to planInstances
10         if there is at least one instance in planInstances then
11             super.checkChildren()
12     endMethod
13 endClass
```

The classifier searches for matches between the elements of the refinement patterns and the models. The search consists of checking the preconditions associated with each pattern starting at the top of the hierarchy and until the leaves are reached or the preconditions at some level fail. That is, progress down the hierarchy can only be made if all the preconditions of a pattern in a node are true. The pseudocode of the accumulator plan shows that there are two preconditions associated with the pattern: i) that an accumulator event occurs in the concrete model (Line 5), and ii) that an abstract atomic action is refined via the accumulator variable associated with the accumulator event (Line 7). When more than one match is found for a pattern precondition, each match is considered as a potential separate instantiation of the pattern (Line 6)–multiple instances may be found as progress is made down the hierarchy when the patterns are more specialised.

When a match is found that meets all the preconditions, a new instance of the pattern is created (Line 8) and the instance is added to the set of plan instances (Line 9). If at least one instance of the pattern is identified (Line 10), the procedure checks the preconditions of the child patterns (Line 11). Note that the *checkChildren* method is defined in the parent class. Then for each child the preconditions are checked and a similar process is started.

After all the patterns have been analysed, and if there is a failure associated with the refinement step, then the classifier invokes the critics analyser with the pattern instances that were identified.

7.1.2.3 Critics Analyser

The role of the analyser is to find ways of overcoming failures associated with a development. Two types of critics have been implemented:

- *Reasoned modelling critics:* these are generic critics which exploit failure at a single level of abstraction in a development. These type of critics were introduced in [69] and their implementation, which is part of this PhD thesis, was reported in [70].

- *Refinement plan critics:* address failures at the level of refinement; and in particular, failures associated with partial or failed instances of refinement patterns.

The general mechanism for both types of critics is the same; i.e. each critic has a set of preconditions that are checked and if successful, automatically generated modelling guidance is provided. The main difference is that refinement plan critics have additional input information; i.e. instances of refinement patterns. Here, the details of refinement plans critics are described.

REMO evaluates the applicability of a critic with respect to a refinement pattern instance, the refinement step and the failed POs. This process consists of the following steps:

1. When the classifier finishes the identification of patterns, if there is a failure associated with the model, the instance is passed to the critics analyser where the associated critics are explored.

2. Then the analyser searches for matches of the critics preconditions. If a precondition being checked is one that identifies different solutions to the same failure (e.g. finding guards that are mutually exclusive with the existing guards of an event), all the matches are part of the set of possible solutions of the failure.

3. For each element of the set of possible solutions the subsequent critic preconditions are evaluated. When all the preconditions have been checked, the analyser verifies which potential solutions were successful in all the preconditions.

4. Then, for each successful critic instance the raw guidance is generated.

5. When the guidance is in the form of a guard or an invariant template, and this template is only partially instantiated, HREMO is used in order to attempt and find the complete instantiation of the guard or invariant template – this step of the process is not implemented yet; however, the integration of REMO and HREMO is part of future work.

6. The analyser collects the results of each successful solution and sends them to the guidance generator.

Note that what is on offer is guidance; whether or not the guidance is accepted is left to the user. Alternatively, where better proof automation is required, the aim in the longer-term is to include a proof planner which will enable REMO to improve upon the level of automation provided by the current Rodin provers (see Chapter 8).

REMO uses the CVC3 SMT solver [11] when a pattern or critic precondition requires proof. This choice of SMT solver follows from an existing OCaml interface developed by Maclean [91]. Currently, only first-order predicates with equality are handled, as well as arithmetic. Thus, the full underlying (set-theoretic) Event-B mathematical notation is not supported. Future work includes taking advantage of SMT solvers for Event-B[2], as well as the use of a proof planner. Furthermore, we may possibly use generic purpose theorem provers.

7.1.2.4 Guidance Generator

The guidance generator takes the raw results from the classifier and the analyser and produces a formatted, ranked (ordered) list of alternative guidance suggestions, which will then be sent to the Rodin plug-in and presented to the user.

Two processes take place here. The first process consists of interpreting the guidance given as an input in order to produce written explanations, while the second process deals with the linking of the guidance and explanation to the Rodin plug-in. Interpreting the guidance produced by the analyser is linked to the critic preconditions of each suggestion. The purpose behind this, is to help the user understand the nature of the guidance, to make a more informed decision about which modification to apply. The guidance produced by the generator will be classified into four categories:

1. **Global guidance:** refers to a global solution that addresses multiple local failures. For instance, the addition of an invariant to the model instead of guards in multiple events or the introduction of an intermediate layer of refinement. This type of guidance is given alongside an explanation of how the local suggestions are used to propose the general one.

2. **Local guidance:** is concerned with a solution that addresses a local failure. For instance, a guard is missing from an event or an event is required in order to overcome deadlock. An explanation of how the failure is overcome with the solution is provided.

3. **Conditional guidance:** refers to solutions that are subject to the fulfilment of certain conditions. For instance, solutions that depend on how variables are prioritised within the context of the modelled system. A list of conditions together with explanations of why these conditions are needed is given with the suggested solution.

[2]See `http://www.cprover.org/SMT-LIB-LSM/`

4. **Partial guidance:** is provided when the problem is identified but the solution is in the form of a partially instantiated template or in the form of a set of possible options. For instance, an invariant template that is not fully instantiated by HREMO. An explanation of the failure together with the partial or set of possible solutions is provided.

The result of the guidance generator is a set of files capturing both the guidance and explanations for each pattern and critic instance. From the user perspective, the guidance will consist of the fragment of Event-B model that is relevant to the pattern and the suggested modification. The natural language explanation is available to user at his/her request.

For instance, in the case of the accumulator plan and in particular the *invariant_speculation* critic, the guidance will contain the current set of invariants plus the new invariant as shown in Figure 7.3.

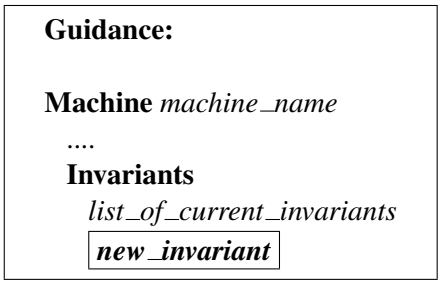


Figure 7.3: Guidance provided by the invariant_speculation critic of the accumulator pattern.

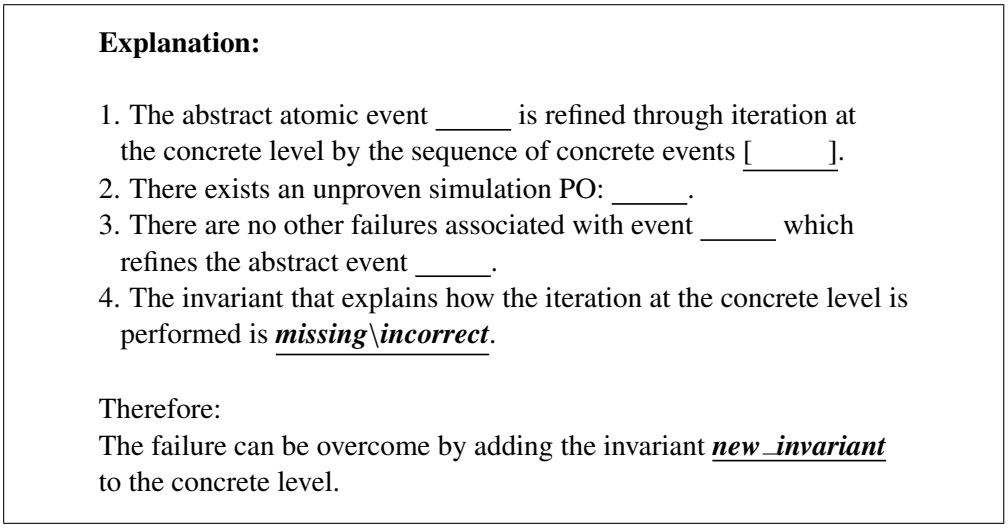


Figure 7.4: Explanation template associated with the invariant_speculation critic of the accumulator pattern.

The explanations are represented as templates, i.e. a natural language explanation of the pattern and critic preconditions, with slots to denote instance specific details. To illustrate,

the proposed template associated with the *invariant_speculation* critic is shown in Figure 7.4 – instantiations of these template are presented in Section 7.2.1.

7.2 Experimental results

A set of Event-B developments were explored in order to identify instances of the more concrete patterns; i.e. the patterns at the leaves of the hierarchy – the developments include models of the Mondex and flash file systems proposed in the verification grand challenge. The summary of this study is given in Table 7.1.

	control refinement						**data refinement**			
Model	**RP1**	**RP2**	**RP3**	**RP4**	**RP5**	**RP6**	**RP7**	**RP8**	**RP9**	**RP10**
Cars on a bridge [3]		2	8				1			
Mondex [22]		1	17	1		4		1	1	2
Flash file system [39]			27		2		2			
Location access ctrl. [3]			1	1		1	3			
Network topology [59]	1		2			2			3	2

Where:

RP1: Control refinement	RP6: Post-decomposition
RP2: Case split	RP7: Set to partition
RP3: Control elaboration	RP8: Partition to function
RP4: Pre-decomposition	RP9: Data extension
RP5: Accumulator	RP10: Data removal

Table 7.1: Refinement pattern analysis of Event-B developments.

The chosen models provided a good set of case studies that allowed us to explore different instances of all the refinement patterns. Moreover, control elaboration is the most common refinement pattern used across the different models. This is expected in a refinement-based formalism like Event-B since this pattern develops existing events without breaking their atomicity or changing the state representation. This is contrary to the other patterns, which introduce new functionalities and modifications to the current representation of the state. As can be observed from Table 7.1, such kind of steps are expected to occur fewer times during the refinement. That is because the initial abstraction should model most of the functionality of the system, and the changes to the state representation are usually performed during the last refinement steps.

The results for each refinement step with the associated refinement patterns are described below – where each refinement step is denoted by the letter **R** and a consecutive number:

1. **Cars on a bridge:** Models a system that controls the flow of cars on a bridge that connects a mainland to an island.

R1: Partitions the number of cars into three variables: cars going towards the island, cars on the island and cars going towards the mainland (RP7).

R2: Introduces traffic lights to control the number of cars allowed in and out of the bridge and island. Splitting events is required in order to handle the traffic lights when the maximum number of cars has been reached (RP2).

R3: Introduces sensors and communication channel variables to model the physical environment in the bridge/island compound. These variables must be handled by the existing events (RP3).

2. **Mondex:** models a smart card that allows the exchange of money between electronic purses.

   R1: The atomic exchange of money between purses is decomposed at the concrete level through a sequence of events that separate the deduction and increment of money from the source purse to the target purse (RP4).

   R2: Two redundant variables that store information about the balance currently engaged in a transaction are removed (RP10).

   R3: Dual states to a transaction are introduced. Splitting of events are required to handle failure of a transaction based on the dual states (RP2), and existing events are modified with details about the new states (RP3).

   R4: Message variables are added that constrain how the communication within the protocol is performed in the existing events (RP3).

   R5: States are associated with purses instead of transactions. This requires the modification of existing events to reference the new states (RP3).

   R6: The global history of transactions is handled in the refinement by local consecutive numbers assigned to each purse involved in a transaction (RP9).

   R7: A redundant variable used to store the history of consecutive numbers is removed (RP10).

   R8: The state sets related to each side of the transaction are handled in the refinement by a status function (RP8).

3. **Flash file system:** Models a flash-based file system that allows a user to read, write and erase information from a flash disk.

   R1: The set of objects is partitioned within the flash file system into files and directories (RP7).

R2: The set of files is partitioned into files being read and files being written (RP7). A function is also introduced to represent the content of files which must be handled by the existing events (RP3).

R3: The concept of users and permissions to files and directories is introduced. This restricts the application and adds effects to the existing events (RP3).

R4: Extra information is added about files and directories; e.g. creation date, name, etc. This data must be modified by the existing events (RP3).

R5: The process of writing the content of a file is decomposed to be performed iteratively in the refinement (RP5).

R6: The process of reading the content of a file is decomposed to be performed iteratively in the refinement (RP5).

4. **Location access controller:** Models a system that controls the access to the rooms within a building.

   R1: The concept of communication between locations is added which must be checked by the existing events (RP3).

   R2: The concept of one way doors to communicate from one location to another is introduced. Accessing a location is decompose in order to verify first if the destination connects with the origin (RP4).

   R3: Card readers which read and pass access information to the system are introduced. The cards must be handled by the existing events in order to transfer the current access information (RP3).

   R4: Physical controls to lock and unlock the doors as well as timers are introduced. This refinement requires the use of partitions in order to handle different states of the doors associated with the controls and timers (RP7).

5. **Network topology:** Models the algorithm used in routing tasks in which each node of a network needs to discover and maintain information about the topology of the network.

   R1: The concept of logical network which is linked with the physical network from the abstraction is introduced (RP1).

   R2: A local network associated with each node which is updated through message channels is introduced. This requires new events to transfer the messages when an update to the network takes place (RP6).

R3: The local network and message variables are extended by adding information about the age of the links being updated, where the parity of the age determine the membership to the network (RP9).

R4: The events that handle the updates to the network and the transfer of messages are merged.

R5: The history of messages is made redundant and therefore it is removed from the model (RP10).

R6: A variable that stores the images of the network built by each node in order to verify that they are equal to the actual network is introduced. This requires existing events to handle the new variable (RP3).

R7: The physical network variable is extended by adding information about the age of the links (RP9). This also makes the variable that keeps the current age of each link redundant (RP10).

7.2.1 Flash file system

Consider the Event-B model shown in Figure 7.5. This model is a fragment of a flash-based file system developed in [39] and refers to the fifth refinement mentioned above. The fragment shown in Figure 7.5 deals with the function of writing the content of a file. In the abstract model, the event *writefile* is responsible for writing the content of file *f*, i.e. *wbuffer(f)*, into a storage variable *fcontent* in an atomic step. In the concrete model the content is written one page at a time into a temporary storage *fcont_tmp* (event *w_step*) before being written to the actual storage *fcontent* (event *w_end_ok*). The new events *w_start* and *w_step* are said to refine *skip*, while event *w_end_ok* refines the abstract event *writefile*. Three invariants are specified at the concrete model, the two first invariants specify the type of the new variables *fcont_tmp* and *writing*, while the last invariant specifies a property of the refinement step, that is, that when the writing process starts for a given file, the content of *fcont_tmp* is a subset or is equal to the content of *wbuffer*.

In order to experiment with REMO, the third invariant was removed and the modified model was input into the tool. Figure 7.6 sums up the steps carried out by REMO during the classification of refinement patterns with respect to the flash file model. The branches represent the patterns at each level of the hierarchy while each node represents the result (or matches) of the analysis of the preconditions of each pattern with the model. Each branch terminates either in failure when it is not possible to find a match with a pattern precondition or in the generation of a leaf pattern instance(s) when a match is found with all its preconditions. In the case of the flash file model an instance of the control refinement pattern is found, that is:

Abstract Model:

Variables
fcontent, w_opened_files, wbuffer, file_size,
power_on, dateLastModified
Invariants
w_opened_files ⊆ files
file_size ∈ files → ℕ
power_on ∈ BOOL
fcontent ∈ files → CONTENT
dateLastModified ∈ (files ∪ directories) → DATE
wbuffer ∈ w_opened_files → CONTENT

Event writefile ≙
any f
where
f ∈ w_opened_files
power_on = TRUE
then
fcontent(f) := wbuffer(f)
dateLastModified(f) := nowdate
file_size(f) := card(wbuffer(f))
end

Concrete Model:

Variables
fcontent, w_opened_files, wbuffer, file_size, power_on, dateLastModified, writing, fcont_tmp
Invariants
writing ⊆ w_opened_files
fcont_tmp ∈ writing → CONTENT
∀ f· f∈ writing ⇒ fcont_tmp(f) ⊆ wbuffer(f)

Event w_start ≙
any f
where
f ∈ w_opened_files
f ∉ writing
power_on = TRUE
then
writing := writing∪{f}
fcont_tmp(f) := ∅
end

Event w_step ≙
any f i data
where
power_on = TRUE
f ∈ writing
i ∈ ℕ
data ∈ DATA
i↦ data ∈ wbuffer(f)
i ∉ dom(fcont_tmp(f))
then
fcont_tmp(f) :=
fcont_tmp(f)∪{i↦data}
end

Event w_end_ok ≙
refines writefile
any f
where
f ∈ writing
dom(fcont_tmp(f))=dom(wbuffer(f))
power_on = TRUE
then
fcontent(f) := fcont_tmp(f)
dateLastModified(f) := nowdate
file_size(f) := card(fcont_tmp(f))
writing := writing \ {f}
fcont_tmp := {f} ⩤ fcont_tmp
end

Figure 7.5: Fragment of the Event-B model of a flash file system developed in [39].

Control_Refinement({*fcont_tmp,writing*}).

This is because a set of new variables is identified at the concrete level in the model; i.e. ({*fcont_tmp,writing*}); however, at the same level of the patterns hierarchy, the data refinement pattern branch fails because all the variables from the abstract level are preserved by the refinement (see Appendix A.2 to refer to the preconditions of each refinement pattern). As a result of the previous analysis, the pattern matching continues only for the children patterns of the control refinement pattern. At the following level an instance of the event atomicity decomposition pattern is found; i.e.:

Event_Atomicity_Decomposition({*fcont_tmp,writing*}, *writeFile, w_end_ok,* {*w_step,w_start*}, {}),

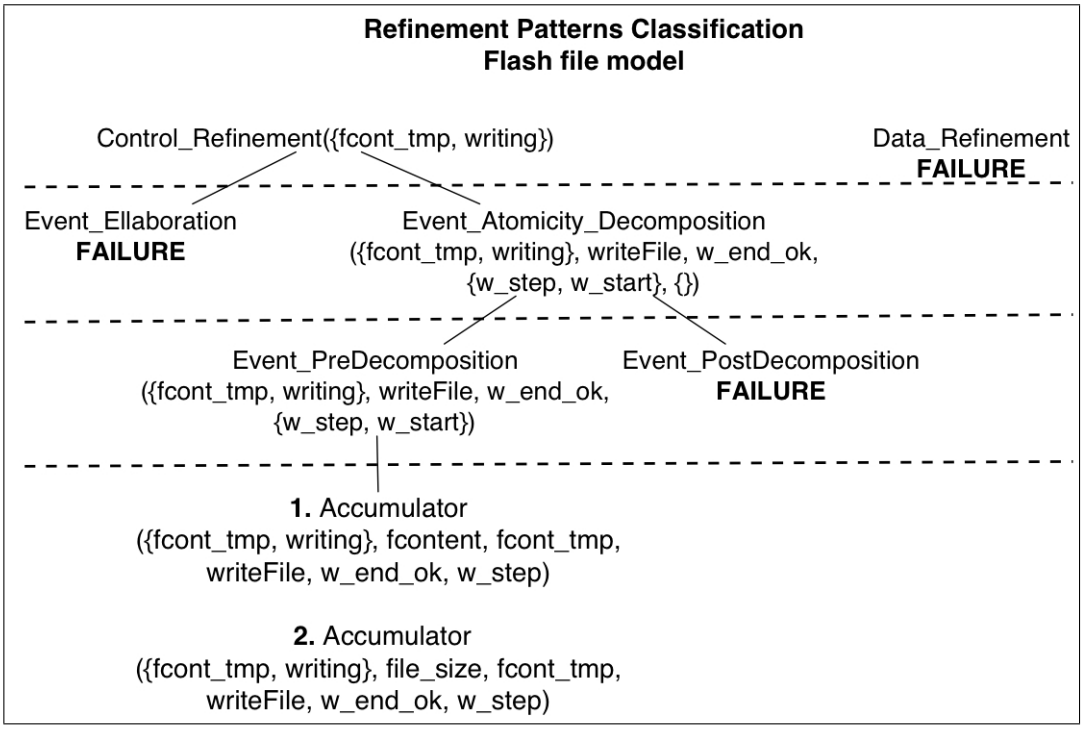


Figure 7.6: Refinement pattern classification in the flash file model.

where the arguments represent: the set of new variables identified by the parent instance, an abstract event (*writeFile*) and the event that refines it (*w_end_ok*), a set of new events that precedes the refined event ({*w_step,w_start*}), and a set of new events that succeeds it (which is empty for this instance). The event elaboration branch fails since the are not events whose refinement is independent of predecessors or successors (which is a precondition of the pattern). The analysis continues only for the children patterns of the event atomicity decomposition pattern, for which REMO finds an instance of the event pre-decomposition pattern since there is a set of new events that precedes the refined event. The instance is:

Event_PreDecomposition({*fcont_tmp,writing*}, *writeFile, w_end_ok,* {*w_step,w_start*}).

At the same level in the hierarchy, the event post-decomposition pattern fails since the refinement of event *writeFile* does not have any successors. The analysis follows with the children of the event pre-decomposition pattern for which two instances of the accumulator pattern are found; these are:

1. *Accumulator*({*fcont_tmp,writing*}, *fcontent, fcont_tmp, writeFile, w_end_ok, w_step*)
2. *Accumulator*({*fcont_tmp,writing*}, *file_size, fcont_tmp, writeFile, w_end_ok, w_step*)

each instance is related to the same accumulator variable and accumulator event; however, they refer to different abstract assignments to variables *fcontent* and *file_size*.

As it stands, the flash file model generates two failed POs associated with the refined event. The failures, shown in Figure 7.7, are produced because it is not possible to prove that the content of the temporal storage variable *fcont_tmp* is equal to the content of the abstract variable *wbuffer* when the refined event is enabled. The failures are identified by REMO and consequently the critics analyser component is invoked. There are five critics associated with the accumulator refinement plan as presented in Section 6.2.1. The preconditions of these critics are analysed against the failed POs, the refinement step and the pattern instances. Table 7.2 summarises the result of this analysis.

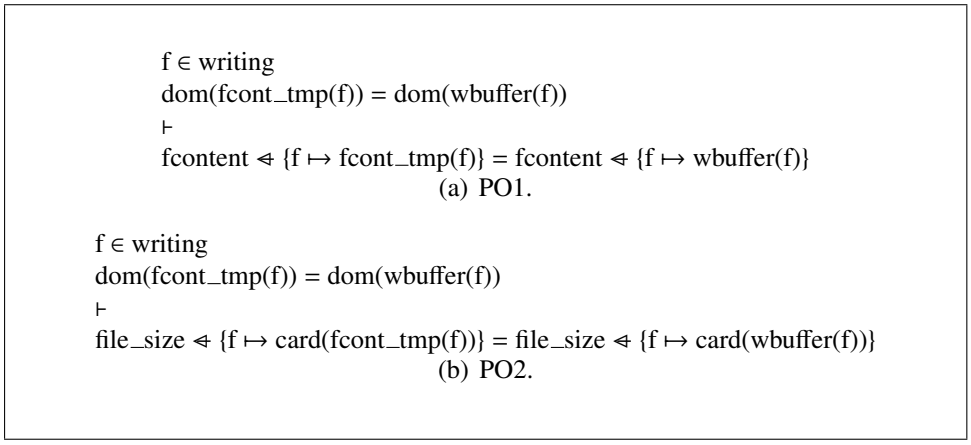


Figure 7.7: Flash file system fifth refinement: Set of failed POs.

Critic	Preconditions result
initialisation_speculation	P1✔, P2✗
loopGuard_speculation	P1✔, P2✗
postGuard_speculation	P1✔, P2✔, P3✗
invariant_speculation	P1✔, P2✔, P3✔, P4✔
accumulator_speculation	P1✗

Table 7.2: Accumulator critics analysis result.

As can be observed there is only one successful critic, the *invariant_speculation critic*, the other critics fail as follows: the *initialisation_speculation* critic fails in precondition 2 because the accumulator event is enabled; the *loopGuard_speculation* critic fails in precondition 2 because a guard with the required shape is identified in the accumulator event; the *postGuard_speculation* critic fails in precondition 3 because a guard with the required shape is identified in the refined event; and the *accumulator_speculation* critic fails in precondition 1 because the refinement step is not an instance of the control elaboration pattern. On the other hand, the successful critic is instantiated as follows:

P1. Two instances of the accumulator pattern are identified in the refinement step.

P2. Two failed SIM POs associated with the refined event and which fit the SIM PO pattern are identified (Figures 7.7(a) and 7.7(b)).

P3. The post-accumulator guard is instantiated to *dom(fcont_tmp(f)) = dom(wbuffer(f))* which is compatible with the guard pattern $G_{=}(\overline{W}, \overline{V_k})$.

P4. The accumulator invariant is missing in the refinement step.

The guidance associated with the critic suggests that an invariant with shape $H_1 \Rightarrow F_{\preceq}(\overline{W}, \overline{V_k})$ must be added to the concrete model. Instantiating the invariant template yields:

$$f \in \text{writing} \Rightarrow F_{\subseteq}(\text{fcont\_tmp, writing, wbuffer})$$

where the containment relationship $\preceq$ is instantiated to $\subseteq$ since the variables in the invariant represent sets (functions fcont_tmp and wbuffer are seen as sets of pairs).

At this point, the guidance is only partially instantiated; however, HREMO can be used to attempt to complete the invariant instantiation. Thus, by using the partially instantiated invariant provided by the *invariant_speculation* critic and the failed POs, HREMO is set to only develop conjectures about the following concepts:

$$\begin{aligned} \textit{core concepts} &= \{\textit{writing, fcont\_tmp, wbuffer, fcontent, file\_size}\} \\ \textit{non-core concepts} &= \{\} \end{aligned}$$

Note that no non-core concepts are identified. Furthermore, the partially instantiated invariant suggests a relationship between sets; therefore, the selected PRs are:

$$\textit{production rules} = \{\textit{compose, disjunct, negate, exists}\}.$$

Finally, only implication conjectures are set to be formed.

After HR is run, two conjectures that match the invariant template are identified:

C1: $\forall A,B,C,D \cdot \text{writing}(A,B) \Rightarrow \text{wbuffer}(A,B,C,D)$

C2: $\forall A,B,C,D \cdot \text{writing}(A,B) \Rightarrow \neg\text{fcont\_tmp}(A,B,C,D) \vee \text{wbuffer}(A,B,C,D)$

where A denotes the state (or step within the simulation trace), B denotes an element of the carrier set OBJECT, C denotes an integer and D denotes an element of the carrier set DATA. The second conjecture discharges the failed POs; therefore, it is suggested as the missing invariant to the user.

The guidance currently reported by REMO is in the form of the partially instantiated invariant since HREMO and REMO are not integrated yet. The partial guidance associated with the flash file system is shown in Figure 7.8(a). On the other hand, the guidance that would be provided via the integration with HREMO is shown in Figure 7.8(b). Note that the invariant is given in the format generated by HREMO since a formal translation to Event-B is yet to be implemented.

The explanation associated with the partial guidance is presented in Figure 7.9. Note that the explanation includes a description of the partial fragment of the invariant.

Guidance:

Concrete model
...
Invariants
writing ⊆ w_opened_files
fcont_tmp ∈ writing → CONTENT
f ∈ writing ⇒ $F_{\subseteq}$(fcont_tmp, writing, wbuffer)

(a) Guidance provided by REMO.

Guidance:

Concrete model
...
Invariants
writing ⊆ w_opened_files
fcont_tmp ∈ writing → CONTENT
∀a,b,c,d · writing(a,b) ⇒ ¬fcont_tmp(a,b,c,d) ∨ wbuffer(a,b,c,d)

(b) Guidance that would be generated with HRemo integration.

Figure 7.8: Guidance associated with the flash file system development.

Explanation:

1. The abstract atomic event ***writeFile*** is refined through iteration at the concrete level by the sequence of concrete events ***w_start, w_step and w_end_ok***.
2. There exists an unproven simulation PO: ***w_end_ok/act1/SIM***.
3. There are no other failures associated with event ***w_end_ok*** which refines the abstract event ***writeFile***.
4. The invariant that explains how the iteration at the concrete level is performed is ***missing***.

Therefore:
The failure can be overcome by adding an invariant with the shape ***f ∈ writing ⇒ $F_{\subseteq}$(fcont_tmp, writing, wbuffer)*** to the concrete level. Where the variables *fcont_tmp, writing* and *wbuffer* are related via an order relation of type ⊆.

Figure 7.9: Explanation provided by REMO for the flash file system.

7.2.2 Mondex

The model shown in Figure 7.10 corresponds to a fragment of refinement three of the Mondex development mentioned above. The refinement step consists of the introduction of dual states to a transaction, and in particular, the fragment shown in Figure 7.10 refers to the

refinement of the event that handles the failure of a transaction. At the abstract model, the event *TransferFail* marks a transaction failed by changing it from the *pending* state to the *recover* state. At the concrete level, the event is split into events *Abortepv1* and *Abortepa1*. The *Abortepv1* event handles failure at the target side of a transaction by changing its state from *epv* (expecting value) to *abortepv*, while the *Abortepa1* event handles failure at the source side by changing the state of the transaction from *epa* (expecting acknowledgement) to *abortepa*.

ABSTRACT MODEL:

Variables
bal purse trans from
to am idle pending
recover ended

Event TransferFail $\widehat{=}$
any t
where
t ∈ pending
then
recover := recover ∪ {t}
pending := pending \ {t}
end

CONCRETE MODEL:

Variables
bal purse trans from to am
idleF epr epa abortepr abortepa
endF idleT epv abortepv endT
Invariants
epv∩(epa∪abortepa)⊆pending
epa∩(epv∪abortepv)⊆pending

Event Abortepv1 $\widehat{=}$
refines TransferFail
any t
where
t ∈ epv
then
abortepv := abortepv∪{t}
epv := epv\{t}
end

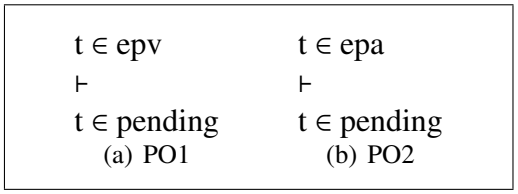
Event Abortepa1 $\widehat{=}$
refines TransferFail
any t
where
t ∈ epa
then
abortepa := abortepa∪{t}
epa := epa\{t}
end

Figure 7.10: Mondex development - Refinement of transfer failure.

As illustrated with the flash file model in the previous section, the classification of refinement patterns is carried out by REMO for the third refinement of the Mondex development– note that the analysis presented here will only refer to the fragment of the model shown in Figure 7.10. REMO identifies an instance of the *case split* refinement pattern which is instantiated as follows:

Case_Split({*idleF,epr,epa,abortepr,abortepa,endF,idleT,epv,abortepv,endT*}, *TransferFail*, {*Abortepv1,Abortepa1*}).

where the first argument represents the set of new variables in the refinement step, the second argument represents an abstract event and the set of evens that refine it are represented by the third argument. As it stands, the model has two failed POs associated with the split events. The failures are shown in Figure 7.11.



(a) PO1 (b) PO2

Figure 7.11: Mondex development third refinement: Set of failed POs.

The critics analyser component is then invoked by REMO which identifies the *guard_speculation* critic as successful. The critic is instantiated as follows:

P1. A case split refinement pattern is identified.

P2. The guard strengthening PO pattern fails for the refined events in the case split (Figures 7.11(a) and 7.11(b)).

P3. Event errors associated with the events in the case split are identified by the ProB simulator. The errors are a result of the events *Abortepv1* and *Abortepa1* being enabled at the concrete level while event *TransferFail* is disabled in the abstraction. This is because the guard $t \in pending$ cannot be satisfied in either case – regarding the execution of event *Abortepv1* the transaction is in the *idle* state, while event *Abortepa1* the transaction is in the *ended* state. Instance traces of these errors are presented below. The traces show the sequence of executed events at both levels of the refinement with their respective parameters. Furthermore, the '-' symbol is used in the abstract trace when an event in the concrete model refines *skip*, while the '✗' symbol is placed when the error occurs:

- Error trace associated with event Abortepv1:
  Abstract: ⟨start(5,p1,p2,t1), -, -, ✗⟩
  Concrete: ⟨start(5,p1,p2,t1), startFrom(p1,t1), startTo(p2,t1), abortepv1(t1)⟩
- Error trace associated with event Abortepa1:
  Abstract: ⟨start(5,p5,p1,t10), - , - , deduct(t10,p5,5), increase(5,p5,p1,t10), ✗⟩
  Concrete: ⟨start(5,p5,p1,t10), startFrom(p5,t10), startTo(p1,t10),
  deduct(5,p5,t10), increase(5,p5,p1,t10), abortepa1(t10)⟩

The guidance associated with the critic suggests adding a guard to the refined events with the shape $H_i(\overline{W})$; i.e.:

$$H(idleF, epr, epa, abortepr, abortepa, endF, idleT, epv, abortepv, endT)$$

That is, the guard is a predicate associated with the set of new variables.

Again, at this point the guidance is only partially instantiated. HREMO is then invoked in an attempt to find the complete instantiation of the missing guards. The selected core and non-core concepts, based on the partially instantiated guard and the failed POs are:

$$\begin{aligned} \textit{core concepts} &= \{\textit{idleF,epr,epa,abortepr,abortepa,endF,idleT,epv,} \\ &\quad \textit{abortepv,endT,pending}\} \\ \textit{non-core concepts} &= \{\} \end{aligned}$$

Furthermore, the variables associated with the partially instantiated guard represent sets; therefore, the selected PRs are:

$$\textit{production rules} = \{\textit{compose, disjunct, negate, exists}\}.$$

Finally, as the search is for a guard, and there are failed GRD POs associated with the events, both equivalence and implication conjectures are set to be formed and the selection focuses on conjectures with the shapes '$\textit{good} \Leftrightarrow \phi$' and '$\phi \Rightarrow \textit{pending}$', where *good* refers to the steps of the trace that do not have errors associated, ϕ is the potential missing guard, and *pending* refers to the abstract guard–in HR the membership to a set; e.g. '$t \in \textit{pending}$' can be represented by just naming the set, in this case *pending*).

Because the invariant is also missing from the model shown in Figure 7.10, it is not possible to ensure if the missing guard has been identified by inspecting the state of the failed POs after the potential guard(s) has been added to the event. Therefore, all conjectures that match the partially instantiated guard template are selected as candidate guards and presented to the user who makes the final decision. This is an example of partial guidance, as introduced in Section 7.1.2.4, for which the solution is presented as a set of possible options. No conjectures with the shape '$\textit{good} \Leftrightarrow \phi$' are generated; however, five conjectures with the shape '$\phi \Rightarrow \textit{pending}$' are identified:

C1: $\forall$A,B · epv(A,B) $\wedge$ (epa(A,B) $\vee$ abortepa(A,B)) $\Rightarrow$ pending(A,B)
C2: $\forall$A,B · epv(A,B) $\wedge$ abortepa(A,B) $\Rightarrow$ pending(A,B)
C3: $\forall$A,B · epa(A,B) $\wedge$ abortepv(A,B) $\wedge$ good(A) $\Rightarrow$ pending(A,B)
C4: $\forall$A,B · epv(A,B) $\wedge$ good(A) $\Rightarrow$ pending(A,B)
C5: $\forall$A,B · epa(A,B) $\wedge$ epv(A,B) $\Rightarrow$ pending(A,B)

where A refers to a state (or step in the simulation trace) and B refers to a transaction. Note that conjectures *C3* and *C4* contain the concept *good* within the potential guards. This show us that the guards hold when there are no error states; therefore, we also consider them as potential guards.

The guidance currently provided by REMO in the case of the Mondex development is shown in Figure 7.12(a). This is in the form of an explanation about the cause of the failure. This is because without the integration with HREMO there are no partially or completed guard

templates provided. The guidance that would be provided via HREMO is shown in Figure 7.12(b) – for simplicity the guards have been converted into Event-B language; however, this translation is not currently automatic.

Note that the potential guards denote the left hand side of the selected conjectures and that '*t*' refers to the parametrised transaction in the concrete events (Figure 7.10). Through manual inspection, the missing guards are identified to be **G2** for event *Abortepv1* and **G3** for event *Abortepa1*. It is important to note that currently there is not support in Rodin for the formulation of guards; therefore, although not exact, the guidance identified through HREMO focuses the user on a small set of possible solutions.

Explanation:

1. A case split has been detected between abstract event ***TransferFail*** and concrete events ***Abortepv1 and Abortepa1***.
2. There exists ***two*** unproven guard strengthening POs: ***Abortepv1/grd1/GRD*** and ***Abortepa1/grd1/GRD***.
3. There are event errors associated with the guards of the split events: *events* ***Abortepv1*** and ***Abortepa1*** *are enabled when the guard of the abstract event* ***TransferFail*** *cannot be satisfied.*

Therefore:
The failures can be overcome by modifying the guards of events ***Abortepv1 and Abortepa1***.

(a) Partially instantiated explanation provided by REMO.

A list of potential guards are:

G1: t ∈ (epv ∩ (epa ∪ abortepa))
G2: t ∈ (epv ∩ abortepa)
G3: t ∈ (epa ∩ abortepv)
G4: t ∈ epv
G5: t ∈ (epa ∩ epv)

(b) Potential guards that would be generated with HRemo integration.

Figure 7.12: Guidance and explanation associated with the Mondex development.

7.3 Summary

This chapter introduced REMO, a prototype tool that implements the refinement plans approach. REMO uses the refinement plans to classify the patterns of refinement used in a development as well as it attempts to overcome failures associated with a pattern instance

through the critics mechanism. Currently, the tool partially implements the hierarchy of patterns so that four of the leaf patterns can be identified; that is: case split, accumulator, set to partition and partition to function patterns. Also, it implements some of the critics associated with them. Although REMO is still at the prototype stage, we have demonstrated through experimental results the general procedure and the current guidance provided to the user. Moreover, we have also demonstrated how partial guidance can be fully instantiated via the integration of REMO and HREMO.

Chapter 8

Conclusions and Future Work

8.1 Summary

Refinement is a powerful technique that addresses the complexity of the design process through incremental steps. This thesis has developed tools and techniques to assist in the development of refinement-based formal models, in particular, refinement of Event-B models. Two complementary techniques have been developed with this goal in mind.

First, building upon HR, animation, and proof-failure analysis, this thesis presented HREMO– an automatic approach to invariant discovery of formal models. HR is a system that implements Automated Theory Formation (ATF), a machine learning technique that builds theories about domains through the inspection of examples that describe the domain, the creation of new concepts via the use of Productions Rules (PRs), and the identification of conjectures that relate the new and old concepts. The approach to automatic invariant discovery developed here uses HR to form theories about models that fail to verify because of the absence of required invariants. This is achieved by:

- using simulation traces, generated by ProB in the case of Event-B, to obtain a set of examples that describe the state of the system,
- syntactically analysing the failures associated with the model in order to assign a higher priority to the development of conjectures of concepts appearing in them, as well as to select relevant PRs to be used during the theory formation process, and
- guiding the selection of interesting conjectures by focusing on the prioritised concepts and by applying a set of filters, such as selecting the most general conjectures and conjectures that only discharge the failed POs.

The final outcome is a set of selected conjectures which represent candidate invariants for the input model.

Second, building upon common patterns of refinement and common patterns of failure, this thesis presents REMO – a tool that automates refinement plans, a technique which provides automatic modelling guidance for users of posit-and-prove style of refinement when a model fails to verify but is close to a known pattern of refinement. Refinement plans use automated analysis of refinement failure at the level of refinements and POs in order to focus the search for modelling guidance. This is achieved by:

- identifying the patterns of refinement that occur in a development,
- using a critics-style mechanism when a refinement step (partially) matches a pattern of refinement but the step has some associated failures, and
- providing guidance as how to overcome the failure when it matches a common pattern of failure associated with the refinement pattern; i.e. when all the preconditions of an associated critic hold.

The guidance provided can be in the form of a complete solution or a partially instantiated template. Experimental results have shown that the HREMO theory formation tool can be used to attempt the fully instantiation of the otherwise partially instantiated guidance.

8.2 Evaluation

8.2.1 Strengths

8.2.1.1 HREMO

As pointed out by the VSI manifesto [61], one of the key aspects for the adoption of formal methods by industry is the ability to support formal verification via automated tools. This thesis has shown the power of HREMO to automatically generate invariants of Event-B models, a process that currently is performed manually. Evidence of this is that HREMO successfully automated the generation of invariants of a re-constructed version of the Mondex development, which required various iterations of manual PO failure analysis in the original development by Butler and Yadav [22]. Although, the automatically discovered invariants are not mathematically challenging, they are numerous and usually represent an obstacle for the wide adoption of formal methodologies.

While HREMO has only been applied to the Event-B formalism, the approach is generic and can be applied to other formalisms. This has been illustrated in this thesis by showing how the technique would be applied to the Z specification of a vending machine system. Furthermore, the generality of the approach also applies to the types of invariants that can be generated. That is, thanks to the use of the general purpose PRs provided by HR, and

to their iterative application, HREMO provides flexibility in the type of invariants that can be discovered; i.e. invariant discovery is not limited to pre-defined templates.

Contrary to existing approaches, e.g. [44], HREMO does not require the user to provide any configuration details in order to focus invariant discovery on interesting properties. This is done automatically by the tool which works in the background without user input. Furthermore, as HREMO identifies the invariants by inspecting the simulation traces rather than by analysing the code, the discovered invariants will always depend entirely on the behaviour of the system and never on the way the model is written or its particular representation in different languages.

8.2.1.2 REMO

The refinement plans mechanism is a novel approach to the application of planning within the context of refinement. Compared to other pattern-based approaches [66, 67, 6, 60], refinement plans benefit from their capability of exploiting partial pattern matching. This inherit flexibility of refinement plans means that REMO is capable of providing the user with insights about their developments, and failures associated with them, without requiring a complete match between an input development and a pattern.

Moreover, the refinement pattern classification identified in this thesis provides an effective mechanism for pattern matching. Since the classification is in the form of a tree structure in which patterns at the bottom of the tree share the characteristics of all the patterns above, it is possible to anticipate that certain patterns will not yield a match so that the system does not spend time and resources exploring them. Furthermore, even if a match with one of the more specialised patterns is not found, i.e. a pattern in the leaves of the tree-structure, a better understanding of the intentions of the user can be obtained by identifying the path within the classification that is followed in the development. Further details about possible directions of work related to this are given in Section 8.3.

8.2.1.3 Integration of HREMO and REMO

The automation achieved by both approaches is largely a result of the productive use of failure which exploits the natural interplay between modelling and reasoning in formal specifications. More specifically, HREMO uses syntactic analysis of failed POs in order to configure HR and to prune the wealth of conjectures it generates. REMO, on the other hand, uses semantic analysis of partial matching and failed POs to guide the understanding of a development and to lead the generation of guidance. This is a key principle of reasoned modelling, which has proven effective as demonstrated through the experiments carried out in this thesis.

Lastly, through the experimental results obtained in this thesis, it can be observed that the integration of HREMO and REMO would provide a framework to support verification of refinement-based formal methods. While HREMO gives flexibility to refinement plans in terms of the guidance that can be generated, i.e. completing the otherwise partially instantiated guidance, REMO can improve the search for invariants in HREMO by providing detailed information about the shape of expected invariants. The experiments have also shown the potential of integrating the approaches in order to discover missing guards. We believe this would provide a firm foundation upon which to further explore techniques that support formal refinement – techniques that suggest design alternatives, whilst removing the burden of proof-failure analysis from the user.

8.2.2 Limitations

Some limitations associated with the work presented in this thesis are outlined below:

The simulation traces require more randomness. Animation is a key aspect of the invariant discovery approach, where the quality of the invariants produced by HREMO strongly depends on the quality of the animation traces. The ProB animator provided good animation traces for most of the experiments; however, ProB is not a test case generator. As a result, some experiments showed a lack of randomness in the generated traces. Specifically, this limitation arose during the analysis of the Mondex and the flash file developments. An alternative way to obtain the required examples is to use the model checker of ProB as a source for the generation of traces instead of the simulator. This would provide more diversity to the examples handed over to HREMO , increasing the chances of discovering the missing invariants.

Current PRs are not tailored for the formal modelling context. As mentioned before and demonstrated through the experimental results, HREMO has proven effective for the discovery of invariants of Event-B models. However, HR was originally developed to form theories about mathematical domains. In this thesis the core of HR was not modified; therefore, only the original PRs were used in the experiments. 7 out of the 22 original PRs were identified to have a correspondence with an Event-B operator. Moreover, the application of some of the suitable PRs is restricted to specific cases. This limits the type of invariants that can be generated by HREMO. We believe there is scope for new PRs that address aspects of the formalism not currently covered; we return to this in Section 8.3.

No external source available to explore failures. The patterns of failures explored in the refinement plans presented in this thesis are limited to those known from personal experience or from recurrent errors mentioned in the literature, such as missing invariants and guards.

However, complex failures are usually not documented or they are difficult to come across. The reason for this is that most developments that are available in the public domain only present the final design. Intermediate and failed attempts are usually not published. In order to identify further failures, it is necessary to form a database of failed and intermediate attempts. A possible way to do this is by incorporating a mechanism into Rodin to store these attempts; however, this is out of the scope of this thesis.

Integration with other formal verification tools. Within Rodin it is not possible to identify if a failure is produced due to errors in the model or because the provers could not handle a proof. Refinement plans address this issue to some extent via the critics mechanism by evaluating characteristics of the model through the preconditions. However, as mentioned before, the available critics are limited to our own experience, so not all cases related to failures may be explored. The integration of HREMO and REMO with a disprover would improve the analysis and effectiveness of both approaches.

8.3 Future work

Some directions for future work have been identified regarding both HREMO and REMO. These are outlined briefly below.

As far as we are aware, ATF techniques had not been investigated within the context of refinement style formal modelling before this thesis. However, we believe the generation of invariants is not necessarily dependent on a specific ATF system. Within ATF there are a number of alternative tools to HR that could be explored. For instance, Montaño-Rivas carried out initial experiments to enable IsaScheme to automatically discover invariant (3.4) by handcrafting a schema to match the structure of the invariant. The main advantage of using IsaScheme in this context would be that it will not generate "non-interesting" invariants, thus bypassing the need for selection heuristics required when using the HR system. However, a disadvantage is that the schema needs to be fine-tuned to match specific invariants; such fine-tuning is only justified if it can be used to invent further invariants. Even with the addition of other schemas, the approach would constrain the type of invariants it is possible to generate: it is not yet known how serious a problem this would be. Additionally, the more schemas there are, the more time the system takes to generate all invariants, so additional schemas may detract from the efficiency of IsaScheme.

Other examples of ATF and MTE (mathematical theory exploration) systems which might find application in this domain include IsaCoSy [74], the CORE system [90] and MATHsAiD [92]. In order to show that ATF techniques in general can be used for automated invariant discovery, the exploration of other ATF systems should be performed.

As mentioned in the previous section, HREMO can be further tailored to the formal modelling context. Some areas in which this can be achieved have been identified:

1. reducing the number of conjectures generated by HR,
2. adding new PRs suitable within formal modelling, and
3. adding support for concepts with sets as parameters.

Initial experiments to address 1 have been carried out. A reduction in the number of generated conjectures can be achieved by constraining the concepts allowed within the theory to only concepts whose set of variables are disjoint. In the first iteration of the Mondex development presented in Section 4.2.1, 7296 conjectures were generated; by applying this constraint, 742 conjectures were created instead. Regarding 2, some examples of suitable new PRs have been highlighted in Section 4.2; for instance, a PR that allows the permutation of columns within a data table. Also, limitations of the current PRs have been identified; for instance, the *numRelation* PR is restricted to concepts of arity 2. Finally, 3 would involve revising each PR in order to handle parameters which represent lists. Addressing these limitations and assessing the development of new PRs are part of the future work agenda.

Currently most of the invariant discovery process within HREMO has been automated; however, heuristics FH4 and FH5 are yet to be implemented. In order to automate these heuristics, tool capabilities that depend on each formalism are required, e.g. a proof obligation generator. The implementation of the heuristics for Event-B would also involve the formal translation of the selected conjectures from the format given by HR to the Event-B language. This translation as well as the automation of heuristics FH4 and FH5 are considered future work. Moreover, as identified through the experimental results, the application of heuristic FH2 may reject interesting conjectures. Additional experiments should be carried out, either to determine a measure to use heuristics selectively; i.e. add the capability to HREMO of disabling heuristics, or to add new heuristics to further optimise the search. A possible new heuristic could be to remove conjectures that only express properties of abstract variables since we are looking for invariants of the refinement step; i.e. the invariants should always involve the state of the concrete level.

As mentioned above, animation is key to our approach, where the quality of the invariants produced by HREMO strongly depends on the quality of the animation traces. A different source for the generation of the animation traces must be explored; candidate sources are the ProB model checker or the use of a test case generator.

Continuing the development of refinement plans and its evaluation using case studies[1] is also an ongoing process. Moreover, experiments with novice users should be carried out

[1]Case studies would mainly be drawn from the DEPLOY project. See `http://www.deploy-project.eu/`.

in order to evaluate the usefulness of the explanations attached to the guidance, and consequently improve these explanations based on the output from the experiments. Furthermore, the link with HREMO will be automated as well as the communication of the results from REMO back to the user. One possible route is via Lopatkin's *transformation patterns* plug-in[2], which allow the transformation of Event-B models via the use of scripts. Furthermore, information about how the events relate to each other should naturally be part of the preconditions of plans and critics. Bendisposto and Leuchel [15], have developed a tool which turns ProB traces into more abstract *flow graphs*, showing the order in which the events may be executed[3]. Support for such "event flow" information should be added in the preconditions of the plans, either as described in [15], or ideally extended with support for infinite systems (in [15] only finite models are supported), which will require theorem proving support.

Refinement plans have been applied for the role of correcting refinements. However, as noted in Section 6.1, refinement plans are yet to be evaluated from the perspective of i) suggesting refinements as in [66, 67, 6, 60]; as well as guiding users in ii) their initial choice of a refinement; and iii) the formulation of abstract models. Abstraction is particularly challenging and we believe it will require interaction with the user. Regarding global analysis, counter example checkers could be used in order to validate the invariants as well as to check for deadlock freedom. Finally, in the longer term we plan to develop a proof planning capability in order to exploit proof methods. This will probably be based upon IsaPlanner [41].

[2]For details see `http://wiki.event-b.org/index.php/Transformation_patterns`.

[3]Hallerstede [57] suggests an approach achieving a similar goal where the user has to add more structure to the model.

Appendix A

Refinement patterns declaration

A.1 Language

This appendix presents the meta-terms used for the declarative representation of refinement patterns. The language is divided into meta-predicates and meta-functions.

A.1.1 Meta-predicates

- **refinesEvent(CE, AE)** : true if event *CE* refines event *AE*.
- **partition(whole, part$_1$, ..., part$_N$)** : true if *part$_1$, ..., part$_N$* partition the set *whole*, i.e.:
  whole = $\bigcup$(part$_1$, ..., part$_N$) $\wedge$
  $\forall$ p$_i$,p$_j$. p$_i$ $\in$ {part$_1$, ..., part$_N$} $\wedge$ p$_j$ $\in$ {part$_1$, ..., part$_N$} $\wedge$ p$_i$ $\neq$ p$_j$ $\Rightarrow$ disjointSets(p$_i$,p$_j$)
- **disjointSets(s$_1$, s$_2$)** : true if the intersection between sets s_1 and s_2 is empty, i.e.:
  s$_1$ $\cap$ s$_2$ = $_\varnothing$
- **extends(v$_1$, v$_2$)** : true if v_1 is a relation whose domain is equals to v_2, i.e.:
  v$_1$ $\in$ v$_2$ $\leftrightarrow$ $_-$
- **isFunction(f,dom,ran)** : true if f is a function with domain *dom* and range *ran*.
- **provable(PO)** : true if *PO* holds, false otherwise.
- **provable(h,PO)** : true if *PO* holds by adding *h* to the set of hypothesis.
- **enables(e_1,e_2)** : true if event e_2 is enabled only after the execution of event e_1.

A.1.2 Meta-functions

- **variables(M)** : returns the set of variables of model *M*.
- **constants(M)** : returns the set of constants of model *M*.
- **events(M)** : returns the set of events of model *M*.
- **actions(E)** : returns the set of actions of event *E*.
- **guards(E)** : returns the set of guards of event *E*.
- **newVariables(AM, CM)** : returns the set of variables that are new in the concrete model *CM*; i.e.:
  newVariables(AM, CM) = {v | v ∈ variables(CM) ∧ v ∉ variables(AM)}
- **newConstants(AM, CM)** : returns the set of constants that are new in the concrete model *CM*; i.e.:
  newConstants(AM, CM) = {c | c ∈ constants(CM) ∧ c ∉ constants(AM)}
- **newEvents(M)** : returns the set of events from model *M* that refine *skip*, i.e.:
  newEvents(CM) = {e | e ∈ events(M) ∧ refinesEvent(e, skip)}
- **predecessor(M)** : contains the pairs of events (e_1, e_2) such that e_1 immediately precedes e_2 in model *M*; i.e. e_2 can be enabled only after the execution of e_1. That is:
  predecessor = {(e_1,e_2) | e_1 ∈ events(M) ∧ e_2 ∈ events(M) ∧ refinesEvent(e_1,skip) ∧ enables(e_1, e_2)}
- **preTClosure(M)** : is the transitive closure of *predecessor(M)*.
- **predecessors(E, M)** : contains all the new events of model *M* that precede event *E*; i.e. the events that must be executed in order to enable event *E*, i.e.:
  predecessors(E, M) = {e_i | (e_i, E) ∈ preTClosure(M)}
- **successor(M)** : contains the pairs of events (e_1, e_2) such that e_1 immediately succeeds e_2 in model *M*; i.e. e_1 can be enabled only after the execution of e_2. That is:
  successor = {(e_1, e_2) | e_1 ∈ events(M) ∧ e_2 ∈ events(M) ∧ refinesEvent(e_1, skip) ∧ enables(e_2, e_1)}
- **sucTClosure(M)** : is the transitive closure of *successor(M)*.
- **successors(E, M)** : contains all the new events of model *M* that succeed event *E*, i.e. the events that are enabled after the execution of event *E*, i.e.:
  successors(E, M) = {e_i | (e_i, E) ∈ sucTClosure(M)}

- **updatedVariables(E, M)** : returns the set of variables that are modified by the event *E*, i.e.:
  updatedVariables(E, M) = {v | v ∈ variables(M) ∧ (F(v):=_) ∈ actions(E) }
- **guardedVariables(E, M)** : returns the set of variables that are conditioned by the event *E*, i.e.:
  guardedVariables(E, M) = {v | v ∈ variables(M) ∧ P(v) ∈ guards(E)}
- **refinedEvents(E, M)** : returns the set of events from model *M* that refine event *E*, i.e.:
  refinedEvents(E, M) = {e | e ∈ events(M) ∧ refinesEvent(e, E)}

A.2 Declarative representation

The declarative representation of the refinement patterns presented in the hierarchy in Section 6.1.2 is provided next. Observe that no reference to PO patterns and critics is provided. This is because so far we have analysed four refinement patterns, one of which was presented in Chapter 6, while the other three are presented in Appendices B, C and D. The arguments marked with '?' are instantiated through the evaluation of the preconditions, while other arguments are instantiated by the preconditions of the parent pattern. Moreover, F($\overline{V}$) is used to denote that an expression over variable(s) $\overline{V}$ occurs.

Control refinement

An instance of this pattern requires:

- the concrete model to have new variables (P1).

Control_Refinement(?newVars)
INPUTS:
  MODELS {AM, CM}
  P_INSTANCE *null*
PRECONDITIONS:
  P1. ?newVars=newVariables(AM, CM) ∧ newVariables(AM, CM)≠ ∅

Event atomicity decomposition

An instance of this pattern requires:

- the refinement step to be an instance of the *control refinement* pattern,
- the concrete model to contain new events (P1),
- the concrete model to contain refined events (P2), and
- the refinement of an event to depend on a sequence of events (P3).

Event_Atomicity_Decomposition(newVars, ?absEvt, ?refEvt, ?pred, ?suc)
INPUTS:
MODELS: {AM, CM}
INSTANCE: *Control_Refinement(newVars)*
PRECONDITIONS:
P1. newEvents(AM, CM) ≠ ∅
P2. refines(?absEvt, ?refEvt, AM, CM)
P3. ?pred=predecessors(refEvt, CM) ∧ ?suc=successors(refEvt, CM) ∧
predecessors(refEvt, CM) ∪ successors(refEvt, CM) ≠ ∅

Event pre-decomposition

An instance of this pattern needs:

- the model to be an instance of the *event atomicity decomposition* pattern,
- a sequence of new events to precede the refined event (P1).

Event_PreDecomposition(newVars, absEvt, refEvt, predecessors)
INPUTS:
MODELS: {AM, CM}
INSTANCE:
Event_Atomicity_Decomposition(newVars, absEvt, refEvt, predecessors, successors)
PRECONDITIONS:
P1. predecessors ≠ ∅

Accumulator decomposition

An instance of this pattern requires:

- the refinement step to be an instance of the *event pre-decomposition* pattern,
- the sequence of predecessors to contain an accumulator event (P1), and
- the refinement of an abstract atomic action to depend on the accumulator variable (P2).

Accumulator(newVars, ?absVar, ?accVar, absEvt, refEvt, ?accEvt)
INPUTS:
MODELS: {AM, CM}
INSTANCE: Event_PreDecomposition(newVars, absEvt, refEvt, predecessors)
PRECONDITIONS:
P1. ?accEvt∈predecessors ∧ ?accVar∈newVars ∧
action(?accVar := ?accVar$\bigoplus \alpha$, ?accEvt)
P2. ?absVar∈V(AM) ∧
action(?absVar := E, absEvt) ∧ action(?absVar := F(?accVar), refEvt)

Event post-decomposition

An instance of this pattern requires:

- the refinement step to be an instance of the *event atomicity decomposition* pattern,
- a sequence of new events to succeed the refined event (P1).

Event_PostDecomposition(newVars, absEvt, refEvt, successors)
  INPUTS:
    MODELS: {AM, CM}
    INSTANCE:
     Event_Atomicity_Decomposition(newVars, absEvt, refEvt, predecessors, successors)
  PRECONDITIONS:
    P1. successors ≠ ∅

Event elaboration

An instance of this pattern requires:

- the refinement step to be an instance of the *control refinement* pattern,
- existing events to be refined (P1), and
- the refined events to remain atomic in the refinement (P2).

Event_Elaboration(newVars, ?absEvt, ?refEvts)
  INPUTS:
    MODELS {AM, CM}
    INSTANCE: *Control_Refinement(newVars)*
  PRECONDITIONS:
    P1. ?absEvt ∈ E(AM) ∧ ?refEvts=refinedEvents(absEvt, AM, CM)
        ∧ refinedEvents(absEvt, AM, CM)≠ ∅
    P2. ∀ e∈?refEvts . predecessors(e, CM) ∪ successors(e, CM) = ∅

Case split

An instance of this pattern requires:

- the refinement step to be an instance of the *event elaboration* pattern,
- the concrete model to refine an event with two or more events (P1).

Case_Split(newVars, absEvt, refEvts)
  INPUTS:
    MODELS {AM, CM}
    INSTANCE *Event_Elaboration(newVars, absEvt, refEvts)*
  PRECONDITIONS:
    P1. |refEvts| ≥ 2

Control elaboration

An instance of this pattern requires:

- the refinement step to be an instance of the *event elaboration* pattern,
- the refined events to be refined by only one event (P1).

Control_Elaboration(newVars, absEvt, refEvt)
  INPUTS:
    MODELS {AM, CM}
    INSTANCE *Event_Elaboration(newVars, absEvt, refEvts)*
  PRECONDITIONS:
    P1. refEvts = {refEvt}

Data refinement

An instance of this pattern requires:

- variables from the abstract model to be removed in the concrete model (P1).

Data_Refinement(?removedVars)
  INPUTS:
    MODELS {AM, CM}
    INSTANCE: *null*
  PRECONDITIONS:
    P1. ?removedVars=variables(AM)\variables(CM) ∧ variables(AM)\variables(CM)≠ ∅

Set to partition

An instance of this pattern requires:

- the refinement step to be an instance of the *data refinement* pattern,
- the concrete model to contain a set of new variables (P1), and
- the concrete model to refine an abstract variable into a partition (P2).

SetToPartition(?absVar, ?partitionVars)
  INPUTS:
    MODELS: {AM, CM}
    INSTANCE: Data_Refinement(removedVars)
  PRECONDITIONS:
    P1. newVariables(AM, CM)≠ ∅
    P2. ?absVar∈removedVars ∧ ?partVars⊆newVariables(AM, CM) ∧ partition(?absVar, ?partVars)

Partition to function

An instance of this pattern requires:

- the refinement step to be an instance of the *data refinement* pattern,
- the abstract model to contain a partition (P1),
- the concrete model to contain new variables (P2),
- the concrete model to contain a partition (P3), and
- the concrete model to refine the abstract partition via a function whose domain is the abstract partition and codomain is the concrete partition (P4).

PartitionToFunction(?absVars, ?function, ?concreteValues)
INPUTS:
MODELS: $\{AM, CM\}$
INSTANCE: Data_Refinement(removedVars)
PRECONDITIONS:
P1. ?absVars $\subseteq$ removedVars $\wedge$ partition(_, ?absVars)
P2. newVariables(AM, CM)$\neq \varnothing$
P3. (?concreteValues$\subseteq$newConstants(AM, CM) $\vee$?concreteValues$\subseteq$newVariables(AM, CM))
$\wedge$ partition(_, concreteValues)
P4. ?function$\in$newVariables(AM, CM) $\wedge$ isFunction(?function, $\bigcup$?absVars, $\bigcup$?concreteValues)

Data extension

An instance of this pattern requires:

- the refinement step to be an instance of the *data refinement* pattern,
- the concrete model to contain new variables (P1), and
- a new variable to extend an abstract variable (P2).

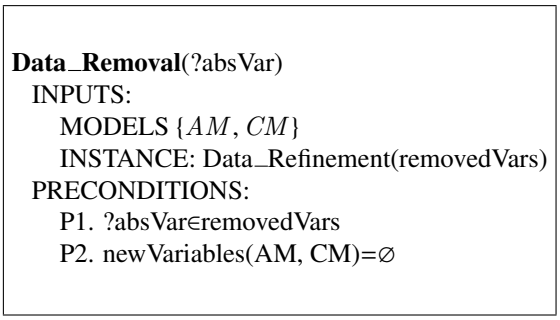
DataExtension(?absVar, ?newVar)
INPUTS:
MODELS $\{AM, CM\}$
INSTANCE: Data_Refinement(removedVars)
PRECONDITIONS:
P1. newVariables(AM, CM)$\neq \varnothing$
P2. ?absVar$\in$removedVars $\wedge$?newVar$\in$newVariables(AM, CM) $\wedge$ extends(?newVar, ?absVar)

Data removal

An instance of this pattern requires:

- the refinement step to be an instance of the *data refinement* pattern,

- abstract variables to be removed in the concrete model (P1),
- without being replaced by new variables (P2).



Appendix B

Case split refinement plan

The modelling and PO patterns of the case split plan are shown in Figures B.1 and B.2, respectively. The key elements in the refinement are:

- The abstract model has an event that is refined in the concrete model.
- A set of new variables $\overline{W}$ are introduced in the concrete model.
- A set of concrete events $case_1$ to $case_N$ that refine the abstract event.
- An invariant, $F(\overline{W}) \Rightarrow G(\overline{V})$, that explains how the guard of the abstract event is implied by the new guard in the concrete model.
- Each case event must preserve the invariant, i.e. Figure B.2(a).
- Each case event must imply the guards of the abstract event, i.e. Figure B.2(b).

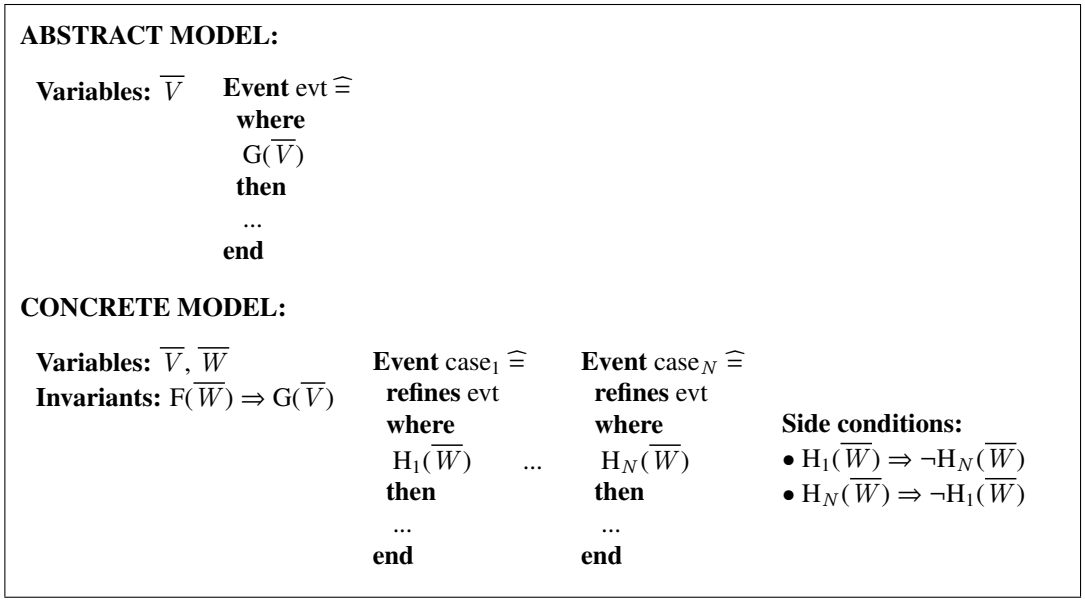


Figure B.1: Case Split plan – Modelling pattern.

$F(\overline{W}) \Rightarrow G(\overline{V})$
$H_i(\overline{W})$
$\vdash$
$[...]F(\overline{W}) \Rightarrow G(\overline{V})$

(a) Case i (Invariant preservation)

$F(\overline{W}) \Rightarrow G(\overline{V})$
$H_i(\overline{W})$
$\vdash$
$[...]G(\overline{V})$

(b) Case i (Guard strengthening)

Figure B.2: Case Split plan – PO patterns.

ABSTRACT MODEL:

Variables
a, b, c
Invariants
a ∈ ℕ
b ∈ ℕ
c ∈ ℕ
n=a+b+c
a=0 ∨ c=0

Event ML_out ≙
refines ML_out
when
a+b< d
c=0
then
a:= a+1
end

CONCRETE MODEL:

Variables
a, b, c, ml_tl, ml_pass
Invariants
ml_tl ∈ Color
ml_pass ∈ 0. . 1
ml_tl=green ⇒ c=0
ml_tl=green ⇒ a+b+c< d

Event ML_out1 ≙
refines ML_out
when
ml_tl=green
a+b+1 < d
then
a:= a+1
ml_pass:= 1
end

Event ML_out2 ≙
refines ML_out
when
ml_tl=green
a+b+1=d
then
a:= a+1
ml_tl:= red
ml_pass:= 1
end

Figure B.3: Instance of the case split refinement pattern – Cars on a bridge model.

We use a fragment of Abrial's cars on a bridge model [3] to illustrate the case split refinement pattern. The example, shown in Figure B.3, consists of an island connected to a mainland by a bridge. The bridge has one lane, and the direction of the traffic is controlled by traffic lights on each side. A maximum number of cars are allowed on the bridge/island, and is denoted by *d*. Variables *a* and *c* denote the numbers of cars travelling towards the island and towards the mainland respectively, while *b* denotes the number of cars on the island. Variables *ml_tl* and *il_tl* represent the traffic lights on the mainland and island, respectively. Event *ML_out* models cars leaving the mainland. Instantiating the key elements listed above from the example instance yields the following:

- Abstract event: *ML_out*.

- Set of new variables $\overline{W}$ = {ml_tl, ml_pass}.
- Set of case events: *ML_out1* and *ML_out2*.
- Invariants that explain the refinement and fit the template *F($\overline{W}$)⇒G($\overline{V}$)*:

$$ml\_tl = green \Rightarrow c = 0$$
$$ml\_tl = green \Rightarrow a + b + c < d$$

Critics

We have identified the following critics associated with the case split refinement pattern:

case_speculation critic: manages the instances of the pattern when a case has not been handled. This critic is applicable iff:

P1. A case split refinement pattern is identified.

P2. The model is not deadlock free.

P3. The case under which the deadlock occurs suggest a set of complementary conditions $\{H_i, ..., H_j\}$ to the current cases within the case split.

Guidance:
Propose new events with complementary guards $\{H_i, ..., H_j\}$. In other words, add events:

Event $case_k \widehat{=}$ **refines** evt **where** H_k **then** ... **end**

where $k \geq i$ and $k \leq j$.

guard_speculation critic: handles the case when a guard of an event in the case split is missing or incorrect. This critic is applicable iff:

P1. A case split refinement pattern is identified.

P2. The guard strengthening PO pattern fails for one or more events in the case split.

P3. There are event errors associated with the guards of the split event(s) that produces the failure – these errors refer to inconsistencies when the abstract and refined events are enabled.

Guidance:
Add guard(s) $H_i(\overline{W})$ to the event(s) associated with the failure.

invariant_speculation critic: manages the instances of the pattern for which the case split invariant is incorrect or missing. This critic is applicable iff:

P1. A case split refinement pattern is identified.

P2. The guard strengthening PO pattern fails for one or more events in the case split.

P3. There are no event errors associated with the split event that produces the failure.

P4. The case split invariant is missing or it is not compatible with the invariant pattern, i.e. F($\overline{W}$)⇒G($\overline{V}$).

Guidance:

An invariant with the shape F($\overline{W}$)⇒G($\overline{V}$) is suggested be added to the concrete model.

Appendix C

Set to partition plan

The modelling and PO patterns of the set to partition plan are shown in Figures C.1 and C.2, respectively. The key elements in the refinement are:

- A set variable V in the abstract level, which is removed at the concrete level.
- Abstract events that add (A_a) and remove (A_r) elements from the set variable.
- A set of new variables $\overline{W}$ = {W$_1$, ..., W$_N$} in the concrete model.
- Refined events that add (C_a) and remove (C_r) elements from one of the new variables.
- Optionally, new or refined events (C_m) that move elements between pairs of new variables.
- An invariant *partition(V, W_1, ..., W_N)* (or a set of equivalent invariants) that specifies that the new variables partition the abstract variable.
- The concrete events must preserve the invariant, i.e. Figures C.2(a), C.2(c) and C.2(e).
- The refined events must imply the guards of the respective abstract event, i.e. Figure C.2(b) and C.2(d).

A fragment of a flash-based file system developed in [39] is used to illustrate an instance of the set to partition refinement pattern. The fragment of the refinement step, shown in Figure C.3, handles the representation of files and directories within the system. In the abstract model, the variable *objects* represents all objects of the system and the event *newobj* adds a new object to the set. In the concrete model the *objects* set is partitioned into the sets *files* and *dirs*, while the abstract event is split into the events *mkdir* and *crt_file*, which create directories and files accordingly – *parent* is a variable used to represent the tree-like structure of objects while *root* is a constant representing the top of the files and directories structure. Instantiating the key elements listed above from the example instance yields the following:

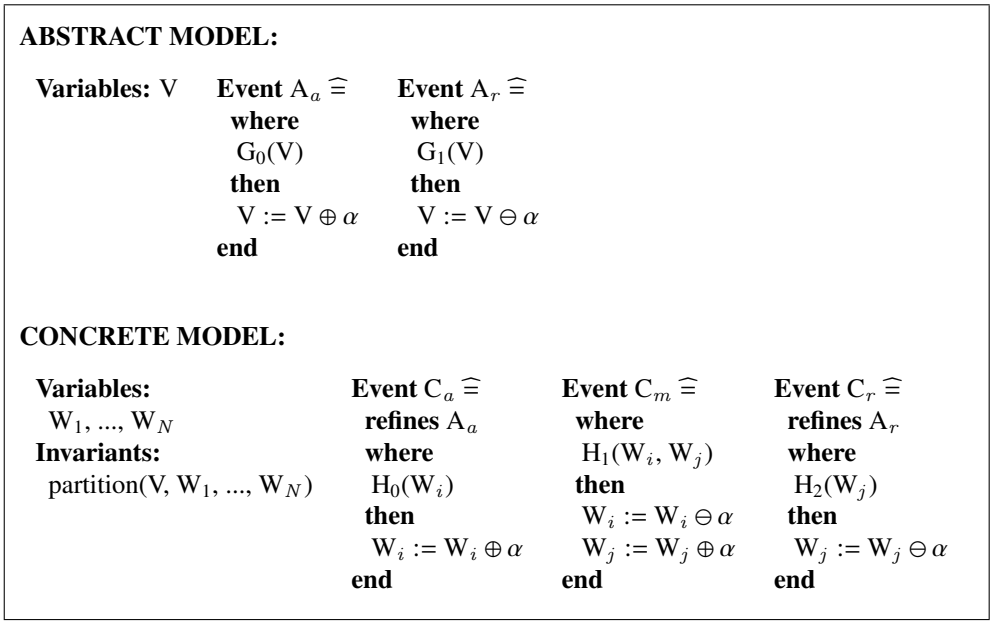


Figure C.1: Schematic representation of the set to partition refinement plan.

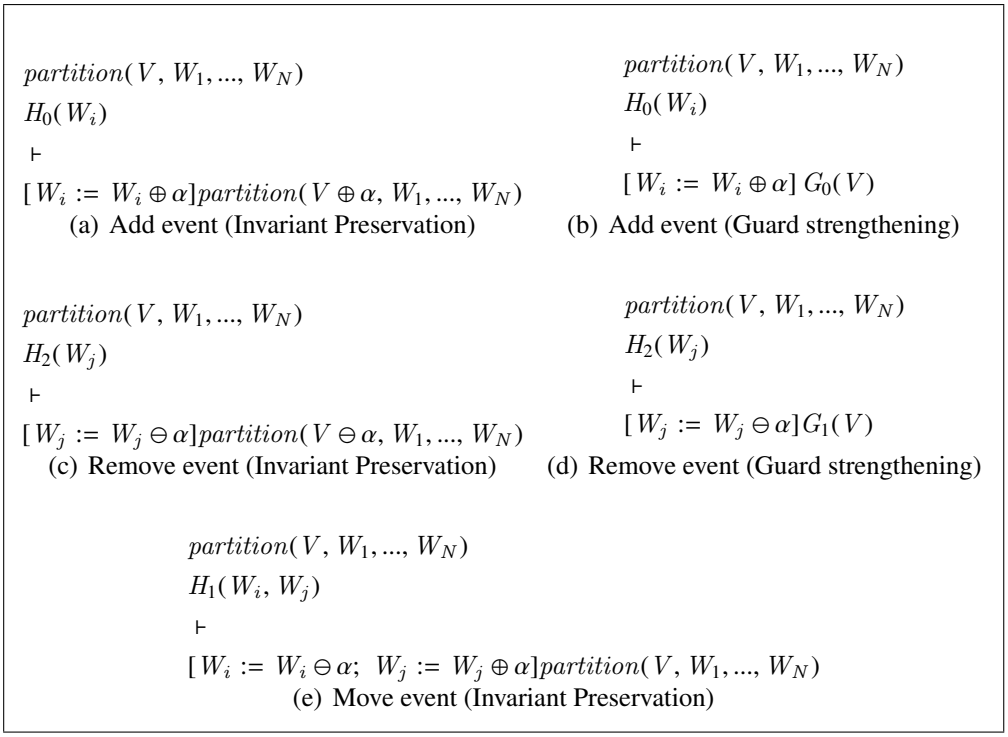


Figure C.2: Set to partition – PO patterns.

- Abstract variable V = *objects*.

- Abstract event A_a = newobj.

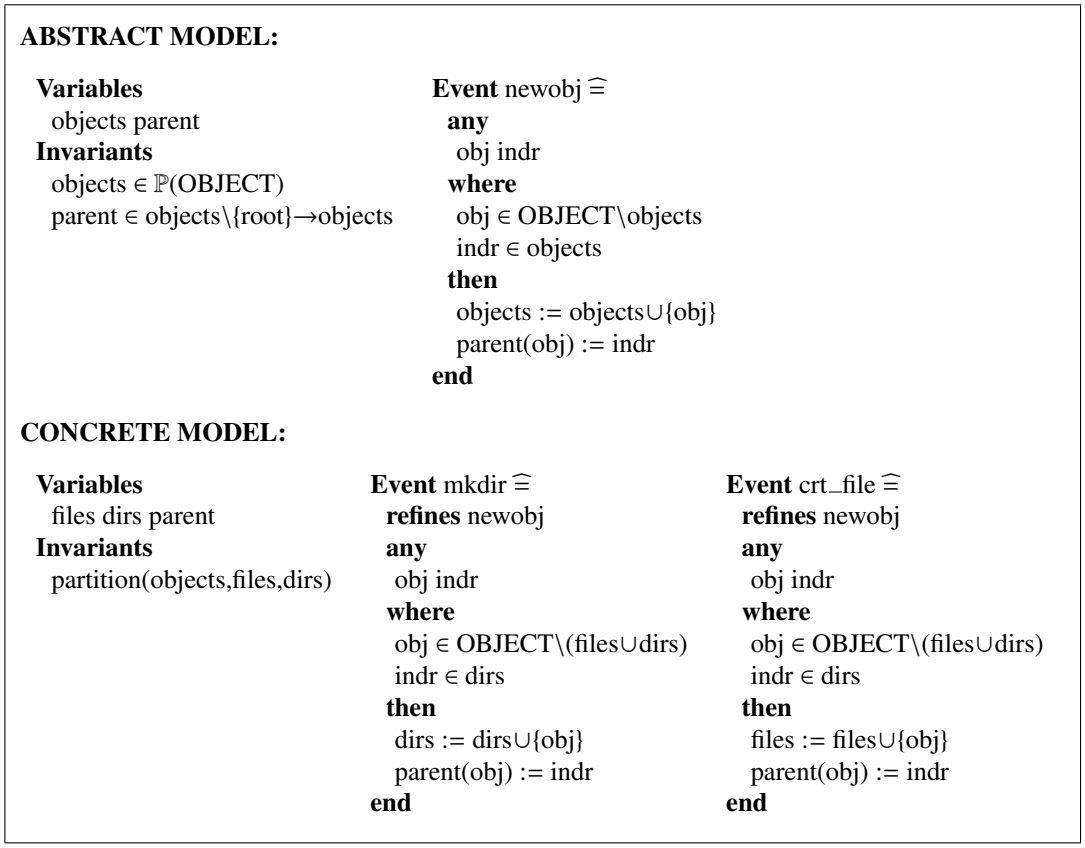


Figure C.3: Fragment instance of the set to partition pattern – Flash file system [39].

- Set of new variables $\overline{W}$ = {*files, dirs*}.
- Refined events C_{a_1} = *mkdir* and C_{a_2} = *crt_file*.
- Invariant specifying the partition: *partition(objects, files, dirs)*.
- Events *mkdir* and *crt_file* refine the abstract event *newobj* by modifying one of the partitioned variables.

Critics

The following critics for the set to partition refinement pattern have been identified:

Merge_partition_events critic: triggers when an event involved in a set to partition pattern is wrongly split, through the partition sets, at the concrete level. This critic is applicable iff:

P1. A set to partition refinement pattern is identified.

P2. An event involved in the pattern is split at the concrete level.

P3. Proof of the guard strengthening PO pattern associated with the split events fails.

P4. The failures can be addressed by merging the cases handled by each event.

Guidance:
Merge the split events into one single event – this involves merging guards and actions, and referring to the union of the partition sets when needed.

Guard_speculation critic: triggers when a guard in the set to partition refinement pattern is missing or incorrect. This critic is applicable iff:

P1. A set to partition refinement pattern is identified.

P2. Proof of the invariant preservation PO pattern associated with an event in the set to partition pattern fails.

P3. The failure can be addressed through the introduction of a guard(s) involving the partition sets being modified in the event.

Guidance:
The correspondent guard(s) H(W_i) or H(W_i,W_j) is suggested to be added to the event.

Event_speculation critic: triggers when a set to partition event requires the introduction of new events to handle the movement of objects through the partition sets. This critic is applicable iff:

P1. A set to partition refinement pattern is identified.

P2. A deadlock occurs at the concrete level.

P3. There are some transitions between partitions sets that do not occur within the model.

Guidance:
A new event is created for each missing transition. Note that the user must identify which transitions are relevant to the model.

Invariant_violation critic: triggers when partitions are not disjoint, i.e. when the same elements are added to different partition sets. This critic is applicable iff:

P1. A set to partition refinement pattern is identified.

P2. The invariant preservation PO pattern fails when the addition of an element(s) α is performed to the partition sets.

Guidance:
The disjointness between the partitions $\overline{W}$ must be ensured. A set of possible solutions should be explored – if more than one solution is applicable, the decision of which option to choose is left to the user:

1. Add a guard $\alpha \notin \mathrm{W}_i$ if *provable(event_guards* $\vdash \alpha \notin W_i$*)* fails.
2. Add action $\mathrm{W}_i := \mathrm{W}_i \ominus \alpha$ if *provable(event_guards* $\vdash \alpha \in W_i$*)* holds.
3. If more than one assignment of the form $\mathrm{W}_i := \mathrm{W}_i \oplus \alpha$ occur, then:
   (a) Remove one (or more) of such assignments.
   (b) Modify the assignments such that all the elements α being added to the partition sets are disjoint with each other.

Example application of the set to partition plan

Consider the abstract and concrete versions of event *copy* shown in Figure C.4. This refinement is associated with the set to partition pattern instance of the flash file system presented in Figure C.3. The purpose of the event is to make a copy of a set of objects (files and/or directories) in a different location within the tree file structure. The root of the objects to be copied is denoted by *obj* while its descendants are represented by *des*. The corresponding new root object is denoted by *nobj*. The union of all objects is represented by *objs* while the corresponding new objects are stored in *nobjs*. The function *corres* expresses the bijection between the two sets. The directory where the objects will be copied is denoted by *to*, and *subparent* and *replica* are the original tree structure and a copy of it, respectively. The event adds the new objects to the set of *files* and *directories*, accordingly, and updates the *parent* function, which maps all objects to their respective parent directory.

As it stands, the event has a failed invariant PO shown in Figure C.5. The invariant violation critic is triggered and is instantiated as follows:

P1. The set to partition refinement pattern is identified.

P2. The failed invariant PO associated with the partition invariant is identified as shown in Figure C.5.

Guidance:
The disjointness between the partitions files and directories must be ensured by:

- Removing action files := files∪corres[objs], or
- Removing action directories := directories∪corres[objs∩directories], or

Event copy $\widehat{=}$
any
obj,des,to,objs,corres,nobjs,nobj,
subparent,replica
where
obj $\in$ objects\{root}
des $\subseteq$ objects
des = (tcl(parent))$^{-1}$[{obj}]
to $\in$ objects
to $\notin$ des $\cup${obj}
objs = des$\cup${obj}
nobjs $\subseteq$ OBJECT\objects
corres $\in$ objs$\rightarrow\!\!\!\!\!\rightarrow$nobjs
nobj = corres(obj)
subparent = des$\lhd$parent
replica = corres^{-1};subparent;corres
then
parent := parent$\cup$replica$\cup${nobj$\mapsto$to}
objects := objects$\cup$nobjs
end

(a) Abstract event.

Event copy $\widehat{=}$
refines copy
any
obj,des,to,objs,corres,nobjs,nobj,
subparent,replica
where
obj $\in$ (files$\cup$directories)\{root}
des $\subseteq$ (files$\cup$directories)
des = (tcl(parent))$^{-1}$[{obj}]
to $\in$ directories
to $\notin$ des$\cup${obj}
objs = des$\cup${obj}
nobjs $\subseteq$ OBJECT\(files$\cup$directories)
corres $\in$ objs$\rightarrow\!\!\!\!\!\rightarrow$nobjs
nobj = corres(obj)
subparent = des$\lhd$parent
replica = corres^{-1};subparent;corres
then
parent := parent$\cup$replica$\cup${nobj$\mapsto$to}
files := files$\cup$corres[objs]
directories := directories$\cup$corres[objs$\cap$directories]
end

(b) Concrete event.

Figure C.4: Event associated with the set to partition instance of the flash file system.

partition(objects,files,directories)
obj $\in$ (files$\cup$directories)\{root}
des $\subseteq$ (files$\cup$directories)
des = (tcl(parent))$^{-1}$[{obj}]
to $\in$ directories
to $\notin$ des$\cup${obj}
objs = des$\cup${obj}
nobjs $\subseteq$ OBJECT\(files$\cup$directories)
corres $\in$ objs $\rightarrow\!\!\!\!\!\rightarrow$ nobjs
nobj = corres(obj)
subparent = des $\lhd$ parent
replica = corres^{-1};subparent;corres
$\vdash$
partition(objects$\cup$nobjs, files$\cup$corres[objs], directories$\cup$corres[objs$\cap$directories])

Figure C.5: Failed invariant PO associated with the copy event.

- Replacing assignment *files* := *files*$\cup$*corres[objs]* for

  files := files$\cup$(corres[objs] \ corres[objs$\cap$directories])

  Note that *corres[objs*$\cap$*directories]* is a subset of *corres[objs]* – the elements added to the *directories* and *files* sets, respectively. Here is where the disjointness of the invariant is violated. Therefore, the assignment to set *files* should be modified.

Observe that options two and three of the critic are not applicable because:

- Adding a guard $\alpha \notin \mathrm{W}_i$ does not work since:

  provable(event_guards ⊢ corres[objs]∉partition(objects,files,directories)), and
  provable(event_guards ⊢ corres[objs∩directories]∉partition(objects,files,directories))

  are both provable since the guards:

  nobjs ⊆ OBJECT\(files∪directories)
  corres ∈ objs ⤖ nobjs

  ensure the freshness of the elements being added to the partition sets.

- Adding an action of the form $\mathrm{W}_j := \mathrm{W}_j \ominus \alpha$ does not work since it has been stated that all the objects being added are fresh.

Appendix D

Partition to function plan

The modelling and PO patterns of the partition to function plan are shown in Figures D.1 and D.2, respectively. The key elements in the refinement are:

- A set of abstract variables, V_1, ..., V_N, that are not preserved in the concrete model.
- A new variable, W, which is a function.
- A set of new values, σ_1, ..., σ_N, which form a partition.
- An invariant, V_k=*F(W,*σ_l*)*, that explains the refinement; i.e. that each abstract variable can be obtained through the function and one of the new values.
- Each refined event must preserve the invariant, i.e. Figures D.2(a), D.2(c) and D.2(e).
- Each refined event must imply the guards of the respective abstract event, i.e. Figure D.2(b), D.2(d) and D.2(f).

An instance of the partition to function refinement pattern was introduced in Chapter 2.2. Remember that in this model, the abstract level handles the stock availability of products through the disjoint sets *available* and *soldout*, whereas at the concrete level the representation is changed to a function (*status*) that maps products to their availability status; i.e. *AVAILABLE* and *SOLDOUT*. Instantiating the key elements of the pattern to the example instance yields:

- Set of abstract variables: {*available, soldout*}.
- New function variable W = *status*.
- Set of new values: {*AVAILABLE, SOLDOUT*}.

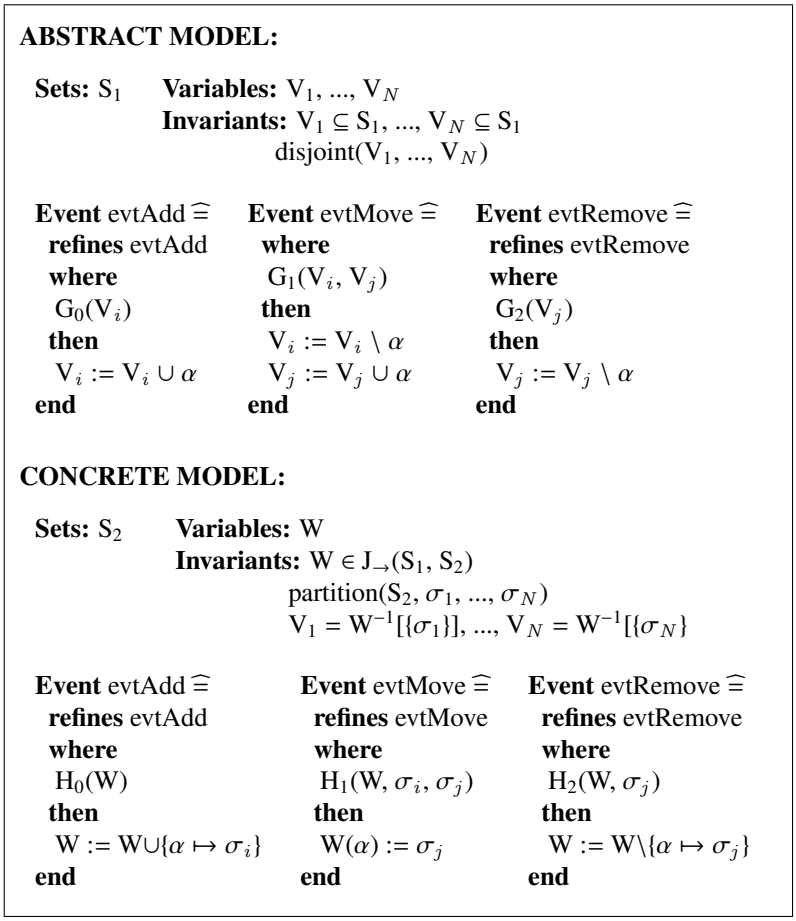


Figure D.1: Partition to function – Schematic representation.

- An invariant with the shape V_k=*F(W,*σ_l*)* is defined for each abstract set; i.e.:

$$available = status^{-1}[\{AVAILABLE\}]$$
$$soldout = status^{-1}[\{SOLDOUT\}]$$

- The respective abstract events are modified at the concrete level by replacing the reference of an abstract set to the concrete function.

Critics

The following critics for the partition to function pattern have been identified:

invariant_speculation critic: handles the case when the invariant is missing or incorrect. This critic is applicable iff:

P1. A partition to function refinement pattern is identified.

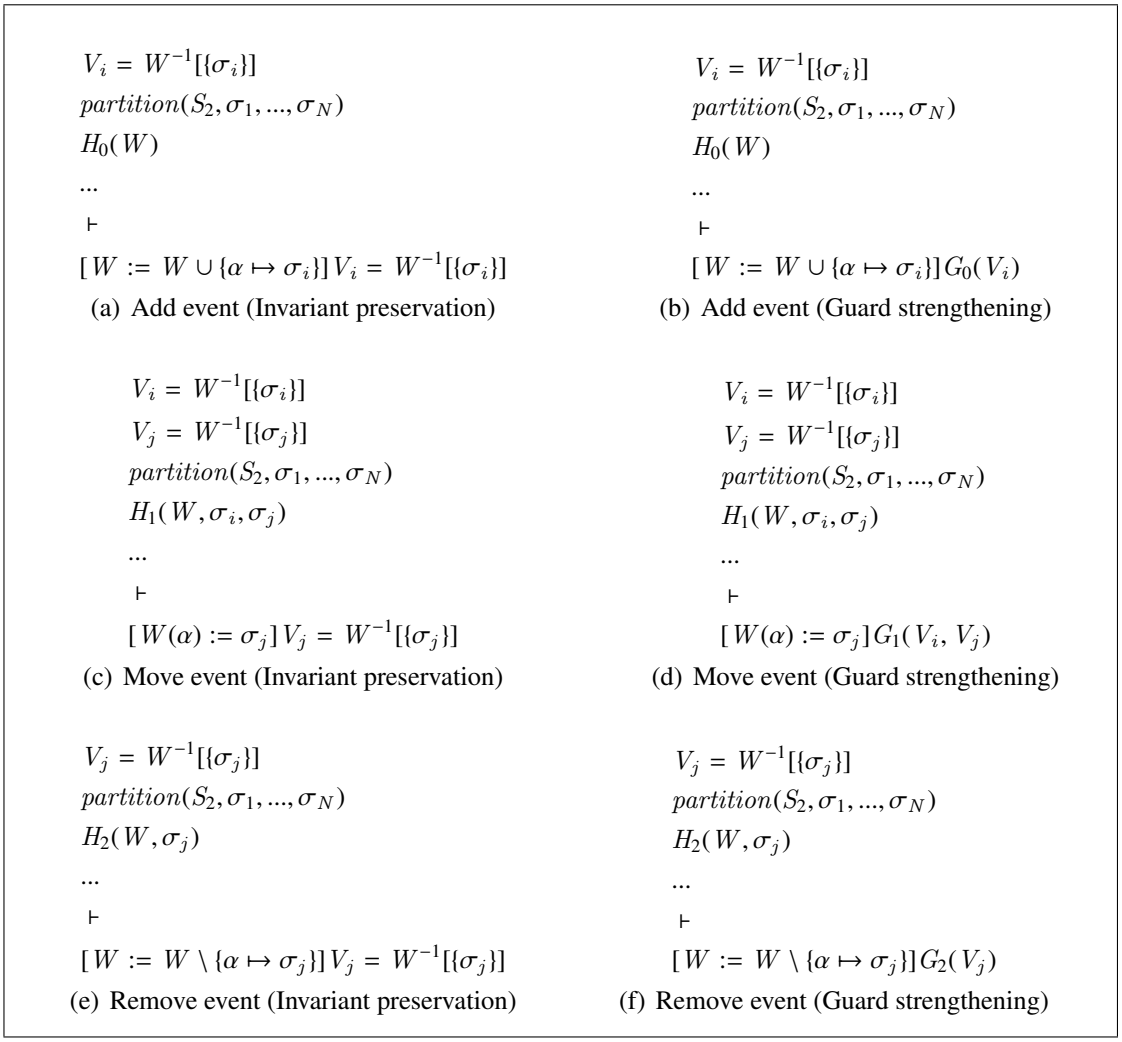


Figure D.2: Partition to function – PO patterns.

P2. The guard strengthening PO associated with an event within the pattern fails. The associated abstract guard involves abstract variable V_j.

P3. The invariant $V_j = W^{-1}[\{\sigma_j\}]$ is missing, is incorrect or it is not compatible with the invariant pattern.

Guidance:

An invariant with the shape $V_j = W^{-1}[\{\sigma_j\}]$ must be added to the concrete model.

invariant_violation critic: handles the case when the relation imposed by an invariant is violated. This critic is applicable iff:

P1. A partition to function refinement pattern is identified.

P2. The invariant preservation PO associated with an event within the pattern fails.

P3. The goal of the failed PO has the shape:

$$V_k \text{ <op> } \alpha = (W\text{<op>}\ \{\alpha \mapsto \sigma_l\})^{-1}[\{\sigma_k\}], \text{ or}$$
$$V_k \text{ <op> } \alpha = W^{-1}[\{\sigma_k\}]$$

That is, an assignment takes place at the abstract level, but at the concrete level the assignment is not consistent or is missing.

P4. The failure can be addressed by adding the assignment:

$$W := W < op > \{\alpha \mapsto \sigma_k\}$$

Guidance:

The event is modified with the assignment that addresses the failure.

Appendix E

HRemo data output examples

ABSTRACT MODEL:

Variables
red_light, green_light
Invariants
red_light ∈ BOOL
green_light ∈ BOOL
green_light=TRUE ⇔ red_light=FALSE

Events
Initialisation
then
red_light := FALSE
green_light := TRUE
end
Event green_to_red ≙
when
green_light = TRUE
then
red_light := TRUE
green_light := FALSE
end
Event red_to_green ≙
when
red_light = TRUE
then
green_light := TRUE
red_light := FALSE
end

CONCRETE MODEL:

Variables
r_light, g_light, amber_light
Invariants
amber_light ∈ BOOL
r_light ∈ BOOL
g_light ∈ BOOL
g_light=TRUE ⇔ green_light=TRUE
r_light=TRUE ∨ amber_light=TRUE ⇔ red_light=TRUE

Events
Initialisation
then
r_light := FALSE
amber_light := FALSE
g_light := TRUE
end
Event amber_to_red ≙
when
amber_light = TRUE
r_light = FALSE
g_light = FALSE
then
r_light := TRUE
amber_light := FALSE
end
Event green_to_amber ≙
refines green_to_red
when
g_light = TRUE
then
amber_light := TRUE
g_light := FALSE
end
Event red_to_redAmber ≙
when
r_light = TRUE
amber_light = FALSE
then
amber_light := TRUE
end
Event redAmber_to_green ≙
refines red_to_green
when
r_light = TRUE
amber_light = TRUE
then
g_light := TRUE
r_light := FALSE
amber_light := FALSE
end

Figure E.1: Event-B model of a traffic light system.

The example shown in Figure E.1 is used to show the output files associated with HREMO. The model consists of a simple traffic light controller. At the abstract level there are only two

possible lights, *green* and *red*. At the concrete level a third light is introduced, i.e. *amber*. This light changes the cycle to: *red, red-amber, red-amber-green, green-amber, red*.

E.1 DTD schema

The DTD schema imposes the format that the simulation trace given as input to HREMO must follow. The DTD schema is shown next.

```
<!ELEMENT domain (domainName, stateConcept,setConcept*,
                  constantConcept*,concept+,po*)>
<!ELEMENT domainName (#PCDATA)>
<!ELEMENT stateConcept (example+)>
<!ELEMENT setConcept (name,example+)>
<!ELEMENT constantConcept (name,parameter+,function?,example+)>
<!ELEMENT concept (name,parameter+,function?,example*)>
<!ATTLIST concept
          abstract (TRUE|FALSE) "TRUE"
          concrete (TRUE|FALSE) "TRUE">
<!ELEMENT name (#PCDATA)>
<!ELEMENT parameter (#PCDATA)>
<!ELEMENT function (dom+,ran+)>
<!ELEMENT dom (#PCDATA)>
<!ELEMENT ran (#PCDATA)>
<!ELEMENT example (#PCDATA)>
<!ELEMENT po (hypothesis*,goal)>
<!ELEMENT hypothesis (formula)>
<!ELEMENT goal (formula)>
<!ELEMENT formula ((formula,bioperator,formula)
          |(bounds?,unioperator,(formula|formulaList))|literal)>
<!ATTLIST formula type (NOTYPE|BINARY|UNARY|LITERAL) "NOTYPE">
<!ELEMENT bioperator EMPTY>
<!ATTLIST bioperator bop (NOBOP|AND|OR|EQV|IMP|EQUAL|NOTEQUAL
|LESSTHAN|LESSTHANEQ|GREATERTHAN|GREATERTHANEQ|IN|NOTIN|SUBSET
|NOTSUBSET|SUBSETEQ|NOTSUBSETEQ|UNION|INTER|BCOMP|FCOMP|OVR
|PLUS|MUL|MAPSTO|RELATION|TOTALREL|SURJECTIVEREL|SURTOTALREL
|PARTIALFUN|TOTALFUN|PARTIALINJ|TOTALINJ|PARTIALSUR|TOTALSUR
|TOTALBIJ|SETMINUS|CARTESIANPRODUCT|DIRECTPRODUCT|PARALLELPRODUCT
|DOMAINRES|DOMAINSUBS|RANGERES|RANGESUBS|UPTO|MINUS|DIVISION|
```

```
MODULO|POWEROF|FUNIMAGE|RELIMAGE) "NOBOP">
<!ELEMENT bounds (#PCDATA)>
<!ELEMENT unioperator EMPTY>
<!ATTLIST unioperator uop (NOUOP|EXISTS|FORALL|FINITE|NOT
|PARTITION|QUNION|QINTER|CSET|SETEXT|CARD|POW|POW1|GENUNION
|GENINTER|DOMAIN|RANGE|FIRSTPROJ|SECONDPROJ|IDENTITY|MIN|MAX|
CONVERSE|UNMINUS) "NOUOP">
<!ELEMENT formulaList (formula+)>
<!ELEMENT literal (#PCDATA)>
```

E.2 Example simulation trace

The simulation trace contains the values of the carrier sets, constants and variables at each step of the simulation. Furthermore, it includes the information of the failed POs, i.e. for each PO it shows the hypotheses and goal.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE domain SYSTEM "hr.dtd">
<domain>
 <domainName>TrafficLight</domainName>
   <stateConcept>
     <example>S0</example>
     <example>S1</example>
     <example>S2</example>
     <example>S3</example>
     <example>S4</example>
     <example>S5</example>
   </stateConcept>
   <setConcept>
     <name>boolean</name>
     <example>TRUE</example>
     <example>FALSE</example>
   </setConcept>
   <concept>
     <name>r_light</name>
     <parameter>state</parameter>
     <parameter>boolean</parameter>
     <example>S0,TRUE</example>
```

```
  <example>S1,TRUE</example>
  <example>S2,FALSE</example>
  <example>S3,FALSE</example>
  <example>S4,TRUE</example>
  <example>S5,TRUE</example>
</concept>
<concept>
  <name>green_light</name>
  <parameter>state</parameter>
  <parameter>boolean</parameter>
  <example>S0,FALSE</example>
  <example>S1,FALSE</example>
  <example>S2,TRUE</example>
  <example>S3,FALSE</example>
  <example>S4,FALSE</example>
  <example>S5,FALSE</example>
</concept>
<concept>
  <name>g_light</name>
  <parameter>state</parameter>
  <parameter>boolean</parameter>
  <example>S0,FALSE</example>
  <example>S1,FALSE</example>
  <example>S2,TRUE</example>
  <example>S3,FALSE</example>
  <example>S4,FALSE</example>
  <example>S5,FALSE</example>
</concept>
<concept>
  <name>amber_light</name>
  <parameter>state</parameter>
  <parameter>boolean</parameter>
  <example>S0,FALSE</example>
  <example>S1,TRUE</example>
  <example>S2,FALSE</example>
  <example>S3,TRUE</example>
  <example>S4,FALSE</example>
```

```
  <example>S5,TRUE</example>
</concept>
<concept>
  <name>red_light</name>
  <parameter>state</parameter>
  <parameter>boolean</parameter>
  <example>S0,TRUE</example>
  <example>S1,TRUE</example>
  <example>S2,FALSE</example>
  <example>S3,TRUE</example>
  <example>S4,TRUE</example>
  <example>S5,TRUE</example>
</concept>
<po>
  <hypothesis>
    <formula type="BINARY">
      <formula type="LITERAL"><literal>g_light</literal></formula>
      <bioperator bop="EQUAL"></bioperator>
      <formula type="LITERAL"><literal>TRUE</literal></formula>
    </formula>
  </hypothesis>
  <goal>
      <formula type="BINARY">
      <formula type="LITERAL"><literal>green_light</literal></formula>
      <bioperator bop="EQUAL"></bioperator>
      <formula type="LITERAL"><literal>TRUE</literal></formula>
    </formula>
  </goal>
</po>
<po>
  <hypothesis>
    <formula type="BINARY">
      <formula type="LITERAL"><literal>r_light</literal></formula>
      <bioperator bop="EQUAL"></bioperator>
      <formula type="LITERAL"><literal>TRUE</literal></formula>
    </formula>
  </hypothesis>
```

```
      <hypothesis>
        <formula type="BINARY">
          <formula type="LITERAL"><literal>amber_light</literal></formula>
          <bioperator bop="EQUAL"></bioperator>
          <formula type="LITERAL"><literal>TRUE</literal></formula>
        </formula>
      </hypothesis>
      <goal>
        <formula type="BINARY">
          <formula type="LITERAL"><literal>red_light</literal></formula>
          <bioperator bop="EQUAL"></bioperator>
          <formula type="LITERAL"><literal>TRUE</literal></formula>
        </formula>
      </goal>
    </po>
</domain>
```

E.3 Example domain file

The domain file contains the same information than the simulation trace, minus the POs, but in the format required by HR. This file represents the input for the theory formation process performed by HR.

```
state
state(A)
prolog:state(@A@)
prover9:state(@A@)
state(S0).
state(S1).
state(S2).
state(S3).
state(S4).
state(S5).

boolean
boolean(A)
prolog:boolean(@A@)
prover9:boolean(@A@)
```

```
boolean(TRUE).
boolean(FALSE).

green_light
green_light(A,B)
green_light(A,B) -> state(A)
green_light(A,B) -> boolean(B)
prolog:green_light(@A@,@B@)
prover9:green_light(@A@,@B@)
green_light(S0,FALSE).
green_light(S1,FALSE).
green_light(S2,TRUE).
green_light(S3,FALSE).
green_light(S4,FALSE).
green_light(S5,FALSE).

red_light
red_light(A,B)
red_light(A,B) -> state(A)
red_light(A,B) -> boolean(B)
prolog:red_light(@A@,@B@)
prover9:red_light(@A@,@B@)
red_light(S0,TRUE).
red_light(S1,TRUE).
red_light(S2,FALSE).
red_light(S3,TRUE).
red_light(S4,TRUE).
red_light(S5,TRUE).

g_light
g_light(A,B)
g_light(A,B) -> state(A)
g_light(A,B) -> boolean(B)
prolog:g_light(@A@,@B@)
prover9:g_light(@A@,@B@)
g_light(S0,FALSE).
g_light(S1,FALSE).
```

```
g_light(S2,TRUE).
g_light(S3,FALSE).
g_light(S4,FALSE).
g_light(S5,FALSE).

r_light
r_light(A,B)
r_light(A,B) -> state(A)
r_light(A,B) -> boolean(B)
prolog:r_light(@A@,@B@)
prover9:r_light(@A@,@B@)
r_light(S0,TRUE).
r_light(S1,TRUE).
r_light(S2,FALSE).
r_light(S3,FALSE).
r_light(S4,TRUE).
r_light(S5,TRUE).

amber_light
amber_light(A,B)
amber_light(A,B) -> state(A)
amber_light(A,B) -> boolean(B)
prolog:amber_light(@A@,@B@)
prover9:amber_light(@A@,@B@)
amber_light(S0,FALSE).
amber_light(S1,TRUE).
amber_light(S2,FALSE).
amber_light(S3,TRUE).
amber_light(S4,FALSE).
amber_light(S5,TRUE).
```

E.4 Example macro file

The macro file contains instructions passed to HR during the theory formation process, i.e. it represents the configuration of HR. The information contained in this file involves: the name of the domain file, the selected PRs, the type of conjectures to be generated, and the number of theory formation steps to be executed.

```
//-------------------------------------------------------
//   domain
//-------------------------------------------------------
clickList("domain_list", "TrafficLight3.hrd", "true");

//-------------------------------------------------------
//   production rules
//-------------------------------------------------------
clickCheckbox("arithmetic_check", "false");
setText("arithmetic_operators_text", "+*-/");
clickCheckbox("disjunct_check", "true");
setText("disjunct_arity_limit_text", "4");
clickCheckbox("negate_check", "true");
clickCheckbox("embed_algebra_check", "false");
clickCheckbox("embed_graph_check", "false");
clickCheckbox("equal_check", "false");
clickCheckbox("entity_disjunct_check", "false");
clickCheckbox("numrelation_check", "false");
clickCheckbox("record_check", "false");
clickCheckbox("size_check", "false");
clickCheckbox("split_check", "false");
clickCheckbox("compose_check", "true");
setText("compose_arity_limit_text", "4");
clickCheckbox("arithmeticb_check", "false");
clickCheckbox("fogpr1_check", "false");
clickCheckbox("linear_constraint_check", "false");
clickCheckbox("parity_check", "false");
clickCheckbox("correlation_check", "false");
clickCheckbox("cross_check", "false");
clickCheckbox("float_arithmetic_check", "false");
clickCheckbox("float_summary_check", "false");
clickCheckbox("forall_check", "false");
clickCheckbox("exists_check", "false");
clickCheckbox("match_check", "false");

//-------------------------------------------------------
//   conjecture making techniques
```

```
//-------------------------------------------------------
clickCheckbox("make_equivalences_from_combination_check", "true");
clickCheckbox("make_implicates_from_subsumes_check", "false");
clickCheckbox("make_implications_from_subsumptions_check", "true");
clickCheckbox("keep_non_exists_check", "true");
clickCheckbox("use_forward_lookahead_check", "true");

//-------------------------------------------------------
//   theory requirements
//-------------------------------------------------------
clickChoice("required_choice", "steps");
setText("required_text", "1000");
setText("complexity_text", "8");

//-------------------------------------------------------
//   Start the theory running
//-------------------------------------------------------
clickButton("start_button");
```

E.5 Example HRemo output

This file contains the output of the invariant discovery process up to filtering heuristic FH3. The output consists of the equivalence, implication and non-existence conjectures associated with the prioritised core and non-core concepts identified in the hypotheses and goals of the POs.

```
PRIORITISED GOAL CORE CONCEPTS
Concept: state(A), boolean(B), green_light(A,B)
2: all A all B  state(A) & boolean(B) & green_light(A,B) <->
    state(A) & boolean(B) & g_light(A,B)
Number of equivalences: 1
0: all A all B  state(A) & boolean(B) & green_light(A,B) ->
    state(A) & boolean(B) & g_light(A,B)
1: all A all B  state(A) & boolean(B) & g_light(A,B) ->
    state(A) & boolean(B) & green_light(A,B)
Number of implications: 2
6: -(exists A exists B state(A) & boolean(B) & green_light(A,B) & red_light(A,B))
86: -(exists A exists T exists R exists U exists E state(A) & boolean(TRUE) &
         green_light(A,TRUE) & red_light(A,TRUE))
110: -(exists A exists T exists R exists U exists E state(A) & boolean(TRUE) &
```

```
           green_light(A,TRUE) & amber_light(A,TRUE))
134: -(exists A exists T exists R exists U exists E state(A) & boolean(TRUE) &
           green_light(A,TRUE) & r_light(A,TRUE))
305: -(exists A exists B state(A) & boolean(B) & green_light(A,B) & amber_light(A,B) &
           ((boolean(B) & red_light(A,B)) | (boolean(B) & r_light(A,B))))
344: -(exists A exists B state(A) & boolean(B) & green_light(A,B) & r_light(A,B) &
           ((boolean(B) & red_light(A,B)) | (boolean(B) & amber_light(A,B))))
507: -(exists A exists B exists T exists R exists U exists E state(A) & boolean(B) &
           g_light(A,B) & boolean(TRUE) & green_light(A,TRUE) &
           ((boolean(B) & red_light(A,B)) | (boolean(B) & r_light(A,B))))
541: -(exists A exists B exists T exists R exists U exists E state(A) & boolean(B) &
           g_light(A,B) & boolean(TRUE) & green_light(A,TRUE) &
           ((boolean(B) & red_light(A,B)) | (boolean(B) & amber_light(A,B))))
579: -(exists A exists B exists T exists R exists U exists E state(A) & boolean(B) &
           g_light(A,B) & boolean(TRUE) & green_light(A,TRUE) &
           ((boolean(B) & r_light(A,B)) | (boolean(B) & amber_light(A,B))))
866: -(exists A exists B exists T exists R exists U exists E state(A) & boolean(B) &
           green_light(A,B) & r_light(A,B) & -(boolean(TRUE) & amber_light(A,TRUE)))
1014: -(exists A exists B exists T exists R exists U exists E state(A) & boolean(B) &
           green_light(A,B) & amber_light(A,B) & -(boolean(TRUE) & r_light(A,TRUE)))
Number of non-exists: 11
Concept: state(A), boolean(B), red_light(A,B)
Number of equivalences: 0
: all A all B  state(A) & boolean(B) & r_light(A,B) & -(green_light(A,B)) ->
  state(A) & boolean(B) & red_light(A,B)
: all A all B  state(A) & boolean(B) & amber_light(A,B) & -(green_light(A,B)) ->
  state(A) & boolean(B) & red_light(A,B)
: all A all B all T all R all U all E  state(A) & boolean(B) & r_light(A,B) &
  boolean(TRUE) & green_light(A,TRUE) -> state(A) & boolean(B) & red_light(A,B)
: all A all B all T all R all U all E  state(A) & boolean(B) & r_light(A,B) &
  -(boolean(TRUE) & amber_light(A,TRUE)) -> state(A) & boolean(B) & red_light(A,B)
: all A all B all T all R all U all E  state(A) & boolean(B) & amber_light(A,B) &
  -(boolean(TRUE) & r_light(A,TRUE)) -> state(A) & boolean(B) & red_light(A,B)
35: all A all B  state(A) & boolean(B) & r_light(A,B) & amber_light(A,B) ->
      state(A) & boolean(B) & red_light(A,B)
43: all A all B  state(A) & boolean(B) & red_light(A,B) ->
      state(A) & ((boolean(B) & r_light(A,B)) | (boolean(B) & amber_light(A,B)))
Number of implications: 7
22: -(exists A exists B state(A) & boolean(B) & red_light(A,B) & g_light(A,B))
Number of non-exists: 1

PRIORITISED GOAL NON-CORE CONCEPTS
Concept: state(A), boolean(TRUE), green_light(A,TRUE)
4: all A all T all R all U all E  state(A) & boolean(TRUE) & green_light(A,TRUE) <->
    state(A) & boolean(TRUE) & g_light(A,TRUE)
```

```
68: all A all T all R all U all E  state(A) & boolean(TRUE) & green_light(A,TRUE) <->
    state(A) & -(boolean(TRUE) & red_light(A,TRUE))
1015: all A all T all R all U all E  state(A) & boolean(TRUE) & green_light(A,TRUE) <->
  state(A) & -(boolean(TRUE) & amber_light(A,TRUE)) & -(boolean(TRUE) & r_light(A,TRUE))
Number of equivalences: 3
: all A all B all T all R all U all E  state(A) & boolean(B) & r_light(A,B) &
  -(boolean(TRUE) & red_light(A,TRUE)) -> state(A) & boolean(TRUE) & green_light(A,TRUE)
88: all A all T all R all U all E  state(A) & boolean(TRUE) & green_light(A,TRUE) ->
    state(A) & -(boolean(TRUE) & amber_light(A,TRUE))
111: all A all T all R all U all E  state(A) & boolean(TRUE) & green_light(A,TRUE) ->
    state(A) & -(boolean(TRUE) & r_light(A,TRUE))
Number of implications: 3
Number of non-exists: 0
Concept: state(A), boolean(TRUE), red_light(A,TRUE)
52: all A all T all R all U all E  state(A) & boolean(TRUE) & red_light(A,TRUE) <->
  state(A) & -(boolean(TRUE) & green_light(A,TRUE))
139: all A all T all R all U all E  state(A) & boolean(TRUE) & red_light(A,TRUE) <->
  state(A) & ((boolean(TRUE) & r_light(A,TRUE)) | (boolean(TRUE) & amber_light(A,TRUE)))
Number of equivalences: 2
: all A all B all T all R all U all E  state(A) & boolean(B) & g_light(A,B) &
  -(boolean(TRUE) & green_light(A,TRUE)) -> state(A) & boolean(TRUE) & red_light(A,TRUE)
3: all A all T all R all U all E  state(A) & boolean(TRUE) & amber_light(A,TRUE) ->
  state(A) & boolean(TRUE) & red_light(A,TRUE)
5: all A all T all R all U all E  state(A) & boolean(TRUE) & r_light(A,TRUE) ->
  state(A) & boolean(TRUE) & red_light(A,TRUE)
954: all A all T all R all U all E  state(A) & -(((boolean(TRUE) & amber_light(A,TRUE)) |
  (boolean(TRUE) & green_light(A,TRUE)))) -> state(A) & boolean(TRUE) & red_light(A,TRUE)
Number of implications: 4
Number of non-exists: 0

PRIORITISED HYPOTHESES CORE CONCEPTS
Concept: state(A), boolean(B), g_light(A,B)
Number of equivalences: 0
Number of implications: 0
365: -(exists A exists B state(A) & boolean(B) & g_light(A,B) & r_light(A,B) &
  amber_light(A,B))
Number of non-exists: 1
Concept: state(A), boolean(B), r_light(A,B)
Number of equivalences: 0
53: all A all B all T all R all U all E  state(A) & boolean(B) & red_light(A,B) &
  boolean(TRUE) & green_light(A,TRUE) -> state(A) & boolean(B) & r_light(A,B)
319: all A all B  state(A) & boolean(B) & red_light(A,B) & -(amber_light(A,B)) ->
  state(A) & boolean(B) & r_light(A,B)
836: all A all B all T all R all U all E  state(A) & boolean(B) & red_light(A,B) &
  -(boolean(TRUE) & amber_light(A,TRUE)) -> state(A) & boolean(B) & r_light(A,B)
```

```
867: all A all B all T all R all U all E  state(A) & boolean(B) & r_light(A,B) ->
  state(A) & ((boolean(B) & green_light(A,B) & boolean(TRUE) & amber_light(A,TRUE)) |
  (boolean(B) & red_light(A,B)))
Number of implications: 4
Number of non-exists: 0
Concept: state(A), boolean(B), amber_light(A,B)
Number of equivalences: 0
54: all A all B all T all R all U all E  state(A) & boolean(B) & red_light(A,B) &
  boolean(TRUE) & green_light(A,TRUE) -> state(A) & boolean(B) & amber_light(A,B)
248: all A all B  state(A) & boolean(B) & red_light(A,B) & -(r_light(A,B)) ->
  state(A) & boolean(B) & amber_light(A,B)
980: all A all B all T all R all U all E state(A) & boolean(B) & red_light(A,B) &
  -(boolean(TRUE) & r_light(A,TRUE)) -> state(A) & boolean(B) & amber_light(A,B)
1020: all A all B all T all R all U all E state(A) & boolean(B) & amber_light(A,B) ->
  state(A) & ((boolean(B) & green_light(A,B) & boolean(TRUE) & r_light(A,TRUE)) |
  (boolean(B) & red_light(A,B)))
Number of implications: 4
Number of non-exists: 0

PRIORITISED HYPOTHESES NON-CORE CONCEPTS
Concept: state(A), boolean(TRUE), amber_light(A,TRUE)
Number of equivalences: 0
Number of implications: 0
Number of non-exists: 0
Concept: state(A), boolean(TRUE), r_light(A,TRUE)
Number of equivalences: 0
955: all A all T all R all U all E state(A) & -(((boolean(TRUE) & amber_light(A,TRUE))
 | (boolean(TRUE) & green_light(A,TRUE)))) -> state(A) & boolean(TRUE) & r_light(A,TRUE)
Number of implications: 1
Number of non-exists: 0
```

Bibliography

[1] Ariane 501 inquiry board. Ariane 5 - Flight 501 failure report. `http://www.di.unito.it/~damiani/ariane5rep.html`, 1996.

[2] J.-R. Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[4] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[5] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[6] J.-R. Abrial and T. S. Hoang. Using design patterns in formal methods: An Event-B approach. In *ICTAC*, volume 5160 of *LNCS*, pages 1–2. Springer, 2008.

[7] J.-R. Abrial, S. A. Schuman, and B. Meyer. *Specification Language*, chapter On the Construction of Programs, pages 343–410. Cambridge Academic Press, 1980.

[8] R.-J. Back. Refinement calculus, Part II: Parallel and reactive programs. In *REX Workshop*, volume 430 of *LNCS*, pages 67–93. Springer, 1989.

[9] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[10] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

[11] C. Barrett and C. Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[12] R. Baumgarten, M. Nika, J. Gow, and S. Colton. Towards the automatic invention of simple mixed reality games. In *AISB*, 2009.

[13] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Developing UPPAAL over 15 years. *Software – Practice and Experience*, 41(2):133–142, 2011.

[14] M. Ben-Ari. *Ada for Software Engineers*. Springer, 2009.

[15] J. Bendisposto and M. Leuschel. Automatic flow analysis for Event-B. In *FASE*, volume 6603 of *LNCS*, pages 50–64. Springer, 2011.

[16] Y. Bertot and P. Casteran. *Interactive theorem proving and program development: Coq'Art: The calculus of inductive constructions*. Springer, 2004.

[17] C. Bolton. Using the Alloy analyzer to verify data refinement in Z. *ENTCS*, 137(2):23–44, 2005.

[18] I. Bratko and M. Grobelnik. Inductive learning applied to program construction and verification. In *AIFIPP*, pages 169–182. North-Holland Publishing Co., 1992.

[19] A. Bundy. The use of explicit plans to guide inductive proofs. In *CADE*, volume 310 of *LNCS*, pages 111–120. Springer, 1988.

[20] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level guidance for mathematical reasoning*. Cambridge University Press, 2005.

[21] M. Butler. Decomposition structures for Event-B. In *IFM*, volume 5423 of *LNCS*, pages 20–38. Springer, 2009.

[22] M. Butler and D. Yadav. An incremental development of the Mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.

[23] B. Calder, P. Feller, and A. Eustace. Value profiling. In *MICRO*, pages 259–269. IEEE Computer Society, 1997.

[24] N. Cataño, T. Wahls, C. Rueda, V. Rivera, and D. Yu. Translating B machines to JML specifications. In *SAC*, pages 1271–1277. ACM, 2012.

[25] A. Cavalcanti and J. Woodcock. ZRC - A refinement calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, 1998.

[26] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO*, volume 4111 of *LNCS*, pages 342–363. Springer, 2005.

[27] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[28] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[29] G. B. Clemmensen and O. N. Oest. Formal specification and development of an Ada compiler - a VDM case study. In *ICSE*, pages 430–440. IEEE Press, 1984.

[30] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[31] S. Colton. Automated puzzle generation. In *AISB*, 2002. Available at `http://www.doc.ic.ac.uk/~sgc/papers/colton_aisb02.pdf`.

[32] S. Colton. *Automated theory formation in pure mathematics*. Springer, 2002.

[33] S. Colton. Creativity versus the perception of creativity in computational systems. In *AAAI*, pages 14–20. AAAI Press, 2008.

[34] S. Colton, A. Bundy, and T. Walsh. Automatic identification of mathematical concepts. In *ICML*, pages 183–190. Morgan Kaufmann, 2000.

[35] S. Colton, A. Bundy, and T. Walsh. Automatic invention of integer sequences. In *AAAI/IAAI*, pages 558–563. AAAI Press / The MIT Press, 2000.

[36] S. Colton and I. Miguel. Constraint generation via automated theory formation. In *CP*, volume 2239 of *LNCS*, pages 575–579. Springer, 2001.

[37] S. Colton and S. Muggleton. Mathematical applications of inductive logic programming. *Machine Learning*, 64(1-3):25–64, 2006.

[38] S. Colton and A. Pease. The TM system for repairing non-theorems. *ENTCS*, 125(3):87–101, 2005.

[39] K. Damchoom. *An incremental refinement approach to a development of a flash-based file system in Event-B*. PhD thesis, University of Southampton, 2010.

[40] L. Dixon. *A proof planning framework for Isabelle*. PhD thesis, University of Edinburgh, 2005.

[41] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *CADE*, volume 2741 of *LNCS*, pages 279–283. Springer, 2003.

[42] E. Dürr and J. van Katwijk. VDM++: A formal specification language for OO designs. In *TOOLS*, pages 63–77. Prentice Hall, 1992.

[43] A. Edmunds, A. Rezazadeh, and M. Butler. Formal modelling for Ada implementations: Tasking Event-B. In *Ada-Europe*, volume 7308 of *LNCS*, pages 119–132. Springer, 2012.

[44] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[45] J. Esparza and K. Heljanko. *Unfoldings - A partial-order approach to model checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

[46] A. S. Fathabadi and M. Butler. Applying Event-B atomicity decomposition to a multi media protocol. In *FMCO*, volume 6286 of *LNCS*, pages 89–104. Springer, 2009.

[47] A. S. Fathabadi, A. Rezazadeh, and M. Butler. Applying atomicity and model decomposition to a space craft system in Event-B. In *NFM*, volume 6617 of *LNCS*, pages 328–342. Springer, 2011.

[48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[49] A. Ghose. Formal methods for requirements engineering. In *ISMSE*, pages 13–16. IEEE Computer Society, 2000.

[50] P. C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, 4(1):28–35, 1960.

[51] R. L. Givan. *Automatically Inferring Properties of Computer Programs*. PhD thesis, MIT, 1996.

[52] P. Godefroid. *Partial-Order methods for the verification of concurrent systems - An approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer, 1996.

[53] M. Gordon. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, volume 35, pages 73–128. Springer, 1987.

[54] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A mechanized logic of computation*. LNCS. Springer, 1980.

[55] G. Grov, A. Ireland, and M. T. Llano. Refinement plans for informed formal design. In *ABZ*, volume 7316 of *LNCS*, pages 208–222. Springer, 2012.

[56] K. Haase. Discovery systems. In *ECAI*, pages 111–120, 1986.

[57] S. Hallerstede. Structured Event-B models and proofs. In *ABZ*, volume 5977 of *LNCS*, pages 273–286. Springer, 2010.

[58] S. Hallerstede, M. Leuschel, and D. Plagge. Refinement-Animation for Event-B - Towards a method of validation. In *ABZ*, volume 5977 of *LNCS*, pages 287–301. Springer, 2010.

[59] T. S. Hoang, D. Basin, H. Kuruma, and J.-R. Abrial. Development of a network topology discovery algorithm. DEPLOY project Repository. Available at http://deploy-eprints.ecs.soton.ac.uk/82/.

[60] T. S. Hoang, A. Fürst, and J.-R. Abrial. Event-B patterns and their tool support. In *SEFM*, pages 210–219. IEEE Computer Society, 2009.

[61] C. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. *ACM Computing Surveys*, 41(4):22:1–22:8, 2009.

[62] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[63] C. A. R. Hoare. Assertions in modern software engineering practice. In *COMPSAC*, pages 459–462. IEEE Computer Society, 2002.

[64] G. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley Professional, 2003.

[65] G. J. Holzmann, R. Joshi, and A. Groce. *25 Years of Model Checking*, chapter New Challenges in Model Checking, pages 65–76. Springer, 2008.

[66] A. Iliasov. Refinement patterns for rapid development of dependable systems. In *EFTS*, page 10. ACM Press, 2007.

[67] A. Iliasov. *Design components*. PhD thesis, University of Newcastle, 2008.

[68] A. Ireland. The use of planning critics in mechanizing inductive proofs. In *LPAR*, volume 624 of *LNCS*, pages 178–189. Springer, 1992.

[69] A. Ireland, G. Grov, and M. Butler. Reasoned modelling critics: Turning failed proofs into modelling guidance. In *ABZ*, volume 5977 of *LNCS*, pages 189–202. Springer, 2010.

[70] A. Ireland, G. Grov, M. T. Llano, and M. Butler. Reasoned modelling critics: Turning failed proofs into modelling guidance. *Science of Computer Programming*, 78(3):293–309, 2013.

[71] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, 2001.

[72] M. Jastram, S. Hallerstede, M. Leuschel, and A. G. Russo. An approach of requirements tracing in formal refinement. In *VSTTE*, volume 6217 of *LNCS*, pages 97–111. Springer, 2010.

[73] R. D. Jeffords and C. L. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *SIGSOFT FSE*, pages 56–69, 1998.

[74] M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *ITP*, volume 6127 of *LNCS*, pages 291–306. Springer, 2010.

[75] C. B. Jones. *Software development: A rigorous approach*. Prentice Hall, 1980.

[76] C. B. Jones. *Systematic software development using VDM*. Prentice Hall, 1990.

[77] C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[78] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable file system. In *VSTTE*, volume 4171 of *LNCS*, pages 49–56. Springer, 2005.

[79] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[80] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[81] I. Lakatos. *Proofs and refutations*. Cambridge University Press, 1976.

[82] K. Lano. Z++, an object-orientated extension to Z. In *Z User Workshop*, pages 151–172. Springer, 1990.

[83] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[84] D. Lenat. Eurisko: A program which learns new heuristics and domain concepts. *Artificial Intelligence*, 21, 1983.

[85] D. B. Lenat. Automated theory formation in mathematics. In *IJCAI*, pages 833–842, 1977.

[86] M. Leuschel and M. Butler. ProB: A model checker for B. In *FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[87] N. G. Leveson and C. S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[88] M. Llano, A. Ireland, and A. Pease. Discovery of invariants through automated theory formation. *Formal Aspects of Computing*, pages 1–47, 2012.

[89] M. T. Llano, A. Ireland, and A. Pease. Discovery of invariants through automated theory formation. In *Refine Workshop*, volume 55 of *EPTCS*, pages 1–19, 2011.

[90] E. Maclean, A. Ireland, L. Dixon, and R. Atkey. Refinement and term synthesis in loop invariant generation. In *WING*, 2009.

[91] E. Maclean, A. Ireland, and G. Grov. The CORE system: Animation and functional correctness of pointer programs. In *ASE*, pages 588–591. IEEE, 2011.

[92] R. McCasland, A. Bundy, and S. Autexier. Automated discovery of inductive theorems. In *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23), pages 135–149. University of Białystok, 2007.

[93] W. McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003.

[94] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[95] K. L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[96] A. Meier, V. Sorge, and S. Colton. Employing theory formation to guide proof planning. In *AISC*, volume 2385 of *LNCS*, pages 275–289. Springer, 2002.

[97] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[98] O. Montano-Rivas, R. L. McCasland, L. Dixon, and A. Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems Applications*, 39(2):1637–1646, 2012.

[99] C. Morgan. *Programming from specifications*. Prentice–Hall, 1990.

[100] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[101] A. Newell and H. A. Simon. The logic theory machine. *Computers and Thought*, 1963.

[102] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.

[103] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[104] A. Pease. *A computational model of Lakatos-style reasoning*. PhD thesis, University of Edinburgh, 2007.

[105] A. Pease, A. Smaill, S. Colton, A. Ireland, M. Llano, R. Ramezani, G. Grov, and M. Guhe. Applying Lakatos-style reasoning to AI problems. In *Thinking Machines and the philosophy of computer science: Concepts and principles*, pages 149–174. IGI Global, 2010.

[106] D. Plagge and M. Leuschel. Validating Z specifications using the ProB animator and model checker. In *IFM*, volume 4591 of *LNCS*, pages 480–500. Springer, 2007.

[107] A. Requet. BART: A tool for automatic refinement. In *ABZ*, volume 5238 of *LNCS*, page 345. Springer, 2008.

[108] G. D. Ritchie and F. K. Hanna. *AM: A case study in AI methodology*. Cambridge University Press, 1990.

[109] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[110] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118. ACM, 1999.

[111] W. Shen. Functional transformations in AI discovery systems. Technical Report CMU-CS-87-117, Computer Science Department. Carnegie Mellon University, 1987.

[112] C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, 2006.

[113] C. F. Snook and R. Harrison. Practitioners' views on the use of formal methods: An industrial survey by structured interview. *Information & Software Technology*, 43(4):275–283, 2001.

[114] M. Spivey. *The Z notation: A reference manual*. Prentice Hall, 1992.

[115] J. Stark. *Proof planning for imperative program development*. PhD thesis, Heriot-Watt University, 2000.

[116] M. Vaziri and G. Holzmann. Automatic detection of invariants in Spin. In *SPIN*, 1998.

[117] J. Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.

[118] J. Woodcock and J. Davies. *Using Z: Specification, refinement and proof*. Prentice-Hall, 1996.

[119] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):19:1–19:36, 2009.