Discovery of Invariants through Automated Theory Formation

Maria Teresa Llano[1], **Andrew Ireland**[1] **and Alison Pease**[1]

[1]School of Mathematical and Computer Science, Heriot-Watt University, UK
{a.ireland,mtl4}@hw.ac.uk

[2]School of Informatics, University of Edinburgh, UK
ggrov@inf.ed.ac.uk

ABSTRACT. Refinement is a powerful mechanism for mastering the complexities that arise when formally modelling systems. Refinement also brings with it additional proof obligations – requiring a developer to discover properties relating to their design decisions. With the goal of reducing this burden, we have investigated how a general purpose automated theory formation tool, HR, can be used to automate the discovery of such properties within the context of the Event-B formal modelling framework. This gave rise to an integrated approach to automated invariant discovery. In addition to formal modelling and automated theory formation, our approach relies upon the simulation of system models as a key input to the invariant discovery process. Moreover we have developed a set of heuristics which, when coupled with automated proof-failure analysis, have enabled us to effectively tailor HR to the needs of Event-B developments. Drawing in part upon case study material from the literature, we have achieved some promising experimental results. In addition, we present initial findings from a comparative study with Daikon – a likely program invariant discovery system, and briefly describe other automated theory formation systems which might be applied to automated invariant discovery. While our focus has been on Event-B, we believe that our approach could be applied more widely to formal modelling frameworks which support simulation.

Keywords: Invariant discovery; Automated theory formation; Formal modelling and refinement.

1 Introduction

By allowing a developer to incrementally introduce design details, refinement provides a powerful mechanism for mastering the complexities that arise when formally modelling systems. This benefit comes with proof obligations (POs) – the task of proving the correctness of each refinement step. Discharging such proof obligations typically requires a developer to supply properties – properties that relate to their design decisions. Ideally, automated tools would support the discovery of such properties, allowing the developer to focus on design decisions rather than analysing failed proof obligations.

With this goal in mind, we have developed a heuristic approach for the automatic discovery of invariants in order to support the formal modelling of systems. Our approach, shown in Figure 1, involves three components:

1. a simulation component that generates system traces;

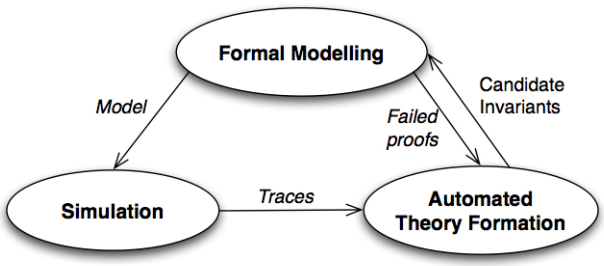

Figure 1: Approach for the automatic discovery of invariants.

2. an Automated Theory Formation (ATF) component that generates conjectures from the analysis of the traces; and
3. a formal modelling component that supports proof and proof failure analysis.

Crucially, proof and proof failure analysis is used to tailor the theory formation component.

From a modelling perspective we have focused on Event-B [Abr10] and the Rodin toolset [ABH+10], in particular we have used the ProB animator plug-in [LB03] for the simulation component. In terms of ATF, we have used a general-purpose system called HR [Col02]. Generating invariants from the analysis of ProB animation traces is an approach analogous to that of the Daikon system [EPG+07]; however, while Daikon is tailored for programming languages here we focus on formal models. We come back to this in §7.

Our investigation involved a series of experiments, drawing upon examples which include Abrial's "Cars on a Bridge" [Abr10] and the Mondex case study by Butler et al. [BY08] . Our initial experiments highlighted the power of HR as a tool for automating the discovery of both system and gluing invariants – system invariants introduce requirements of the system while gluing invariants relate the state of the refined model to the state of the abstract model. However, our experiments also showed significant limitations: i) selecting the right configuration for HR according to the domain at hand, i.e. selection of production rules and the number of theory formation steps needed to generate the missing invariants, and ii) the overwhelming number of conjectures that are generated. This led us to consider how HR could be systematically tailored to provide practical support during an Event-B development. As a result we developed a set of heuristics which are based upon proof-failure analysis. We have implemented these heuristics within a prototype framework. In this paper we present the heuristics, describe the prototype and report on some promising experimental results. Although Event-B is the current focus of our work, we believe our approach can be applied to any refinement style formal modelling framework that supports simulation and that uses proof in order to verify refinement steps.

The remainder of this paper is organised as follows. In §2 we provide background on both Event-B and HR. The application of HR within the context of Event-B is described in §3, along with the limitations highlighted above. In §4 we present our heuristics, and describe their rationale. A description of the prototype implementation is presented in §5. Our experimental results are given in §6, while a comparison with Daikon, and other related

techniques are discussed in §7 along with future work.

2 Background

2.1 Event-B

Event-B promotes an incremental style of formal modelling, where each step of a development is underpinned by formal reasoning. An Event-B development is structured around *models* and *contexts*. A context represents the static parts of a system, i.e. *constants* and *axioms*, while the dynamic parts are represented by models. Models have a state, i.e. *variables*, which are updated via guarded actions, known as *events*, and are constrained by *invariants*.

To illustrate the basic features of a refinement, we draw upon the Mondex development [BY08] – a system that models the transfer of money between electronic purses. Specifically, we focus on the refinement step shown in Figure 2. Note that an event takes the following general form:

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

Moreover, where an abstract event is refined, the keyword **refines** is used to indicate the *refining* event at the concrete level. This is illustrated by the *StartFrom* event in Figure 2 which 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 2, the state of a purse at the abstract level 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*.

It is important to note that the refinement presented in Figure 2 is unprovable as it stands. In order to verify the refinement, invariants are required that relate the concrete and abstract states – these are known as *gluing invariants*. For the parts of the refinement given in Figure 2, the required gluing invariant takes the form:

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

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*. Within the Rodin toolset*, the user is required to supply such gluing invariants. Likewise, invariants relating to state variables within a single model must also be supplied by the user – what we refer to here as *system invariants*. To illustrate, the following disjointness property represents an invariant of the abstract event above:

$$\text{eprP} \cap \text{idleFP} = \emptyset$$

*Rodin provides an Eclipse based platform for Event-B, with a range of modelling and reasoning plug-ins, *e.g.* UML-B [SB06], ProB model checker and animator [LB03], B4free theorem prover (*http://www.b4free.com*).

Abstract Level

Context:
Sets
purseSet

Model:
Variables
idleFP, eprP, epaP, aborteprP, abortepaP,
endFP, idleTP, epvP, abortepvP, endTP
Invariants
idleFP $\subseteq$ purseSet
eprP $\subseteq$ purseSet
epaP $\subseteq$ purseSet
aborteprP $\subseteq$ purseSet
abortepaP $\subseteq$ purseSet
endFP $\subseteq$ purseSet
idleTP $\subseteq$ purseSet
epvP $\subseteq$ purseSet
abortepvP $\subseteq$ purseSet
endTP $\subseteq$ purseSet
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 $\in$ purseSet $\leftrightarrow$ status
Events
...
StartFrom $\widehat{=}$
refines StartFrom
any t, p1
where
p1 $\mapsto$ IDLEF $\in$ statusF
...
then
statusF(p1) := EPR
...
end
...

The above refinement step affects the *context* as well as the *model*. At the abstract level, the state of purses is represented by a set (*purseSet*) of disjoint sets (*idleFP, eprP, epaP, etc*), whereas at the concrete level the representation is changed to a partial function (*statusF*), i.e. a mapping from purses to status (*IDLEF, EPR, EPA, etc*). Note that *partition* is a primitive of Event-B, and is used here to ensure that the constants that denote the status of a purse are distinct.

Figure 2: A refinement step from the Mondex [BY08] development.

This invariant states that a purse cannot be in both states, *epr* and *idleFP*, simultaneously.

From a theoretical perspective such 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.

2.2 Automated theory formation and HR

Automated theory formation (ATF) and mathematical theory exploration (MTE) are closely related fields concerned with the automatic generation of mathematical theories. This includes some of the aspects of mathematical thought which typically theorem provers do not focus on, such as the invention of definitions, theorems, conjectures, problems, examples and axioms. George Polya, mathematician and educator, was an early influence on these fields. He characterised ways in which people reason and problem-solve in mathematics, publishing his insights in four influential books [Pol45, Pol54a, Pol54b, Pol62]. His heuristics for problem solving include "look at simple cases and try to find a pattern", "explore a specific case" and "explore the conditions of the problem"; he also recommended using analogy and scientific induction. These techniques inspired Lenat, who developed one of our earliest examples of a discovery system in mathematics: Automated Mathematician [Len76] and its successor Eurisko [Len83]. Despite subsequent methodological criticism of Lenat's work [RH90], he did show us that it is possible to formalise heuristics for discovery in mathematics, and, more generally, that developers of systems in ATF and MTE can find inspiration in work by historians of mathematics [Kop75], psychologists [Lef72], educators [Lam72] and mathematicians themselves [Had49] (all cited in [Len76]). Further developments in ATF include theory formation programs [SB89, Faj88, Eps88], which reason in the domains of graph theory and plane geometry. More recently, Lakatos's [Lak76] characterisation of ways in which mathematicians respond to counterexamples and use them to evolve concepts, conjectures and proofs form the basis of theory formation systems HRL [Pea07] and TM [CP05]. Although historically ATF has been developed in mathematics domains, it is now being applied to other areas, including biology, and invention of games, puzzles and scene generation. Montaño-Rivas use a scheme-based approach for the invention of concepts and conjectures in their system IsaScheme [MRMB11]. Within MTE, McCasland *et al.* explore what mathematicians mean by "theorem", which is not just a statement derivable from a set of axioms, but carries an inherent value judgement of the import of the statement. McCasland has developed MATHsAiD [MBS06], a system which reasons deductively from axioms to provable statements and then evaluates them to determine whether they merit the title of "theorem". In the Theorema project [BBCJ+05], Buchberger and colleagues have developed a software system which supports all phases of the mathematical exploration cycle: formalisation, proving, solving and computing. Below we consider the HR system in more detail.

Colton's machine learning system HR[†] [Col02] performs descriptive induction 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

[†]HR is named after mathematicians Godfrey Harold Hardy (1877 - 1947) and Srinivasa Aiyangar Ramanujan (1887 - 1920).

of finding a definition for a target concept. Based on his observation that it is possible to gain an understanding of a complex concept by decomposing it via small steps into simpler concepts, Colton defined production rules which take in concepts and make small changes to produce further concepts.

HR constructs a theory by finding examples of objects of interest, inventing new concepts, making plausible statements relating those concepts, evaluating both concepts and statements and, if working in a mathematical domain, proving or disproving the statements. Objects of interest are the entities which a theory discusses. For instance, in number theory the objects of interest are integers, in group theory they are groups, etc. Concepts are either provided by the user (core concepts) or developed by HR (non-core concepts) and have an associated data table (or table of examples). The data table is a function from an object of interest, such as the number 1, or the prime 3, to a truth value or a set of objects.

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. Production rules include:

- The *split* rule: this extracts the list of examples of a concept for which some given parameters hold.
- The *negate* rule: this negates predicates in the new definition.
- The *compose* rule: combines predicates from two old concepts in the new concept.
- The *arithmetic* rule: performs arithmetic operations (+, -, $*$, $\div$) on specified entries of two concepts.
- The *numrelation* rule: performs arithmetic comparisons ($<$, $>$, $\leq$, $\geq$) on specified entries of two concepts.

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.

Thus, the theories HR produces contain concepts which relate the objects of interest; conjectures which relate the concepts; and proofs which explain the conjectures. Theories are constructed via theory formation steps which attempt to construct a new concept and, if successful, formulate conjectures and evaluate the results. HR has been used for a variety of discovery projects, including mathematics and scientific domains (it has been particularly successful in number theory [CBW00] and algebraic domains [MSC02]) and constraint solvers [CM01, PSC$^+$10].

As an example, we show how HR produces the concept of prime numbers and the conjecture that all prime numbers are non-squares. Figure 3 shows the data tables used by HR for the formation of the concept of prime numbers.

In order to generate this concept, HR would take in the concept of divisors ($b|a$ where b is a divisor of a), represented by a data table for a subset of integers (partially shown in

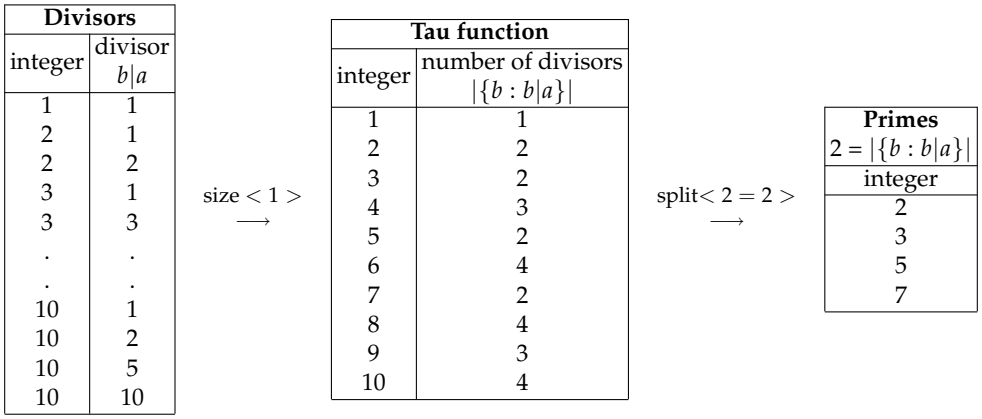

Figure 3: Steps applied by HR to produce the concept of prime numbers.

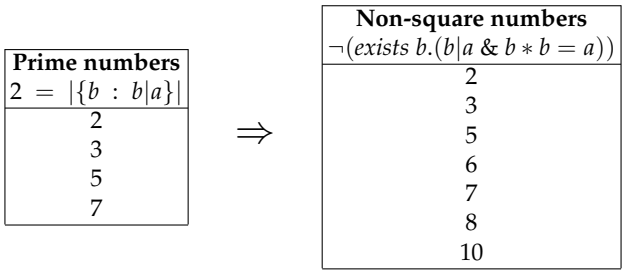

Figure 4: Data tables for the concepts of prime and non-square numbers between 1 and 10.

Figure 3 for integers from 1 to 10). Then, HR would apply the size production rule with the parameterisation $< 1 >$. This means that the number of tuples for each entry in column 1 are 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 a new data table (the table for the concept Tau function). HR then takes in this new concept and applies the split production rule with the parameterisation $< 2 = 2 >$, which means that it produces a new data table consisting of those entries in the previous data table whose value in the second column is 2. This is the concept of a prime number.

After this concept 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 formed previously by HR, the data tables of both the concept of prime numbers and the concept of non-square numbers, shown in Figure 4, are compared.

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:

$$\underbrace{2 = |\{b : b|a\}|}_{\text{prime number}} \Rightarrow \underbrace{\neg(exists\ b.(b|a\ \&\ b * b = a))}_{\text{non-square number}}$$

HR takes input in two files, a domain file which contains the building blocks of the theory (the background concepts and objects of interest), and a macro file which contains instructions for the way in which the theory should be constructed (which production rules are to be used, which arguments they will take, a weighted sum for the interestingness measures, etc.).

3 Automated theory formation for Event-B models with HR

In this section we show how the gluing invariant (1) introduced in the example of §2.1 can be generated through the use of theory formation and, in particular, with the HR system.

3.1 Mondex invariant discovery

HR's input consists of a set of core concepts that describe the domain. With respect to Event-B models, these core concepts are represented by the state of the system, i.e. variables, and by the static information given in the context of the model, i.e. constants and sets. To illustrate, consider again the refinement given in Figure 2. Here the variables *IdleFP, eprP, epaP* etc, the constants *IDLEF, EPR, EPA* etc, and the sets *purseSet* and *status* all represent core concepts. Note that the process of identifying core concepts for a given Event-B refinement step is fully automatic. Details on the automated process are presented in §5. Once identified, HR requires examples to be provided for each of the core concepts. In terms of Event-B, animation (simulation) provides the source of examples. Again drawing upon the Mondex development, Figure 5 illustrates the nature of an Event-B animation trace, where *idleFP* and *statusF* denote abstract and concrete core concept respectively. As mentioned earlier, ProB [LB03] provides the required examples by automatically generating the animation traces directly from the models. However, HR does not work directly with animation traces, it works instead with *data tables*. For each core concept a data table is therefore required. Example data tables are shown in Figure 6 for the core concepts *purseSet*, *status*, *idleFP* and *statusF*. Note that for each *variable* core concepts, e.g. *ideFP* and *statusF*, an extra argument is recorded in the data tables, i.e. the *state* associated with the example.

Given the data tables, HR applies all possible combinations of concepts and production rules in order to generate new concepts and form conjectures. After 433 steps, HR forms the concept that the set *IdleFP* (abstract) is equivalent to the inverse image of the set $IDLEF$ under the function *statusF* (concrete). This discovery involves noticing that data table **idleFP(A,B)** is identical to the rows of data table **statusF(A,B,C)** for which **C** is IDLEF. HR's *split* production rule plays a central role in such discoveries as illustrated in Figure 7. Note that an intermediate output is generated with all tuples of concept *statusF* whose third column matches the parameter *IDLEF*. Since the third column is the same for all tuples of the intermediate concept, this column is removed from the final output concept.

Immediately after the generation of new concepts, HR looks for relationships with other existing concepts. As shown in Figure 8, HR found that the new concept has the same list of examples as concept *idleFP*, which gives rise to the following equivalence conjecture:

$$\forall A, B.((state(A) \wedge purseSet(B) \wedge idleFP(A,B)) \Leftrightarrow (status(IDLEF) \wedge statusF(A,B,IDLEF)))$$

status={IDLEF,EPR,EPA,ABORTEPR,ABORTEPA,ENDF,IDLET,EPV,ABORTEPV,ENDT}
purseSet={purse1,purse2,purse3,purse4,purse5}

	Variables	
state	**idleFP**	**statusF**
S0	-	-
S1	-	-
S2	-	-
S3	-	-
S4	-	-
S5	purse3	purse3↦IDLEF
S6	purse3, purse5	purse3↦IDLEF,purse5↦IDLEF
S7	purse5	purse3↦EPR,purse5↦IDLEF
S8	purse5	purse3↦EPR,purse5↦IDLEF
S9	-	purse3↦EPR,purse5↦EPR
S10	-	purse3↦EPR,purse5↦EPA
⋮	⋮	⋮
S58	-	purse1↦ABORTEPA,purse5↦ENDF
S59	-	purse1↦ABORTEPA

Figure 5: An example animation trace.

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

purseSet(B)
purse1
purse2
purse3
purse4
purse5

status(C)
IDLEF
EPR
EPA
ABORTEPR
ABORTEPA
ENDF
IDLET
EPV
ABORTEPV
ENDT

idleFP(A,B)	
S5	purse3
S6	purse3
S6	purse5
S7	purse5
S8	purse5
S19	purse4
S25	purse5
S29	purse1
S30	purse1
S31	purse1
S38	purse5
S39	purse5
S40	purse5
S44	purse5
S45	purse5
S52	purse5
S53	purse5
S54	purse5

statusF(A,B,C)		
S5	purse3	IDLEF
S6	purse3	IDLEF
S6	purse5	IDLEF
S7	purse3	EPR
S7	purse5	IDLEF
S8	purse3	EPR
S8	purse5	IDLEF
⋮	⋮	⋮
S29	purse1	IDLEF
S29	purse5	ABORTEPR
S30	purse1	IDLEF
S30	purse5	ABORTEPR
S31	purse1	IDLEF
S31	purse5	ABORTEPR
⋮	⋮	⋮
S59	purse1	ABORTEPA

Figure 6: Example HR data tables for core concepts.

Input		
statusF(A,B,C)		
S5	purse3	IDLEF
S6	purse3	IDLEF
S6	purse5	IDLEF
S7	purse3	EPR
S7	purse5	IDLEF
S8	purse3	EPR
S8	purse5	IDLEF
⋮	⋮	⋮
S29	purse1	IDLEF
S29	purse5	ABORTEPR
S30	purse1	IDLEF
S30	purse5	ABORTEPR
S31	purse1	IDLEF
S31	purse5	ABORTEPR
⋮	⋮	⋮
S59	purse1	ABORTEPA

split<3,IDLEF> ⟶

Intermediate		
statusF(A,B,IDLEF)		
S5	purse3	IDLEF
S6	purse3	IDLEF
S6	purse5	IDLEF
S7	purse5	IDLEF
S8	purse5	IDLEF
S19	purse4	IDLEF
S25	purse5	IDLEF
S29	purse1	IDLEF
S30	purse1	IDLEF
S31	purse1	IDLEF
S38	purse5	IDLEF
S39	purse5	IDLEF
S40	purse5	IDLEF
S44	purse5	IDLEF
S45	purse5	IDLEF
S52	purse5	IDLEF
S53	purse5	IDLEF
S54	purse5	IDLEF

⟶

Output	
statusF_IDLEF(A,B)	
S5	purse3
S6	purse3
S6	purse5
S7	purse5
S8	purse5
S19	purse4
S25	purse5
S29	purse1
S30	purse1
S31	purse1
S38	purse5
S39	purse5
S40	purse5
S44	purse5
S45	purse5
S52	purse5
S53	purse5
S54	purse5

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 7: Split rule applied to obtain the concept of purses whose status in function *statusF* is *IDLEF*.

which can be represented in Event-B as (1) — the gluing invariant introduced in the example of §2.1.

3.2 Challenges in applying HR

For the domain of the Mondex system a total of 4545 conjectures was generated after 1000 formation steps. This is a considerable set of conjectures to analyse. In general, using HR for the discovery of invariants presented us with three main challenges:

1. The HR theory formation mechanism consists of an iterative application of production rules over existing and new concepts. 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 formation steps were run.
2. 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* production rule 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

statusF_IDLEF(A,B)	
S5	purse3
S6	purse3
S6	purse5
S7	purse5
S8	purse5
S19	purse4
S25	purse5
S29	purse1
S30	purse1
S31	purse1
S38	purse5
S39	purse5
S40	purse5
S44	purse5
S45	purse5
S52	purse5
S53	purse5
S54	purse5

$\Leftrightarrow$

idleFP(A,B)	
S5	purse3
S6	purse3
S6	purse5
S7	purse5
S8	purse5
S19	purse4
S25	purse5
S29	purse1
S30	purse1
S31	purse1
S38	purse5
S39	purse5
S40	purse5
S44	purse5
S45	purse5
S52	purse5
S53	purse5
S54	purse5

Figure 8: Formed equivalence conjecture.

domain; therefore, a technique was needed in order to perform this selection.

3. Finally, as highlighted in our example, 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 represented interesting invariants of the system. Thus, 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 order to overcome these challenges, we have developed an approach that uses proof failure analysis to guide the search in HR. In the next section, we introduce this approach and illustrate its application, based on our running example from the Mondex case study.

4 Proof failure analysis and HR

In order to use HR, a user must first configure the system for their application domain. This involves the user in selecting production rules and conjecture making techniques, as well as deciding how many steps HR should be run. In the example introduced in §3.1, the application of the split production rule with respect to the concept *statusF*, for the value *IDLEF*, is an informed decision, based upon the user's knowledge of the model. On its own, HR does not have the capability of applying this type of reasoning. Often 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.

Here we have developed a heuristic approach with the aim of automating this trial and error process. Our heuristics exploit the strong interplay between modelling and reasoning in Event-B. In the context of the discovery of invariants through theory formation, we use the feedback provided by failed POs to make decisions about how to configure HR in order to guide the search for invariants. Specifically, our approach consists of analysing the structure of failed POs so that we can automate:

1. the prioritisation in the development of conjectures about specific concepts,
2. the selection of appropriate production rules that increase the possibilities of producing the missing invariants and,
3. the filtering of the final set of conjectures to be analysed as possible candidate invariants.

4.1 Heuristics

Our heuristics constrain the search for invariants by focusing HR on concepts that occur within failed POs. We use two classes of heuristics – those used in configuring HR, i.e. *Configuration Heuristics (CH)*, and those used in selecting conjectures from HR's output, i.e. *Selection Heuristics (SH)*. Below we explain each class of heuristics in turn:

HR configuration heuristics

We use two overall heuristics when configuring HR for a given Event-B refinement:

CH1. *Prioritise core and non-core concepts that occur within the failed POs as follows:.*
Goal concepts: concepts that appear within the goals of the failed POs.
Hypotheses concepts: concepts that appear within the hypotheses of the failed POs.
Other concepts: concepts that do not appear within the failed POs.

CH2. *Select production rules which will give rise to conjectures relating to the concepts occurring within the failed POs, i.e.*
Split rule: *is selected if members of finite sets occur*.
Arithmetic rule: *is selected if there are occurrences of arithmetic operators, e.g.* +,-,*,/.
Numrelation rule: *is selected if there are occurrences of relational operators, e.g.* $>,<,\leq,\geq$.
In addition, 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.

Below we provide the rationale for these heuristics:

- HR uses an agenda mechanism to organise the theory formation steps. The purpose of CH1 is to give higher priority to core and non-core concepts that occur within the failed POs, which means HR will generate related conjectures earlier within the theory formation process by having the prioritised concepts in the top of the agenda. Furthermore, 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. As a result, such concepts are assigned the highest priority in the application of heuristic CH1. The concepts associated to the hypotheses follow in order of interest, while the remaining concepts are given the lesser priority.
- The missing invariants that are required in order to overcome proof failures will typically have strong syntactic similarities with the failed POs. This is the intuition behind CH2, which selects production rules that focus HR's theory formation process on such syntactic similarities.

As will be shown in §6, the empirical evidence we have gathered so far supports our rationale.

Conjecture selection heuristics

In order to prune the set of conjectures generated by HR, we use the following five selection heuristics:

SH1. *Select conjectures that focus on prioritised core and non-core concepts.*

SH2. *Select conjectures where the sets of variables occurring on the left- and right-hand sides are disjoint.*

SH3. *Select only the most general conjectures.*

SH4. *Select conjectures that discharge the failed POs.*

SH5. *Select conjectures that minimise the number of additional proof failures that are introduced.*

The rationale for these heuristics is as follows:

- SH1 initiates the pruning of uninteresting conjectures by selecting those that describe properties about the prioritised core and non-core concepts (as identified by CH1). Furthermore, the selected conjectures should focus purely on the prioritised concepts; this means that we are interested only in equivalence and implication conjectures of the forms:

$$\alpha \Leftrightarrow \beta$$
$$\alpha \Rightarrow \beta$$
$$\beta \Rightarrow \alpha$$

  where α relates to a prioritised core or non-core concept and β to any other concept. All non-existence conjectures associated with the prioritised concepts are selected. Note that this selection criteria still gives rise to a large set of conjectures. However, as explained in the rationale of CH1 in §4.1, in most cases we have identified the missing invariants by focusing first on the concepts associated to the goals of the failed POs. For the selection process the same reasoning is followed and, therefore, heuristics SH1 to SH5 are focused first on conjectures associated to the concepts of the goals identified by the application of CH1. If no candidate invariants are found, or if old failures are still not addressed by the identified invariants, then the selection process starts again from SH1 to SH5 but focused on the conjectures associated with the concepts of the hypotheses.
- SH2 further prunes the set of conjectures by selecting only those that do not use the same variable(s) in both sides of the conjecture. The reason for this is that invariants in Event-B typically express relationships between different variables of the model.
- SH3 is used to eliminate redundancies amongst the set of selected conjectures by removing those that are logically implied by more general conjectures.
- SH4 is used to select candidate invariants which discharge the given failed POs.
- Potentially, overcoming one proof failure via the introduction of missing invariants may give rise to new proof fails. SH5 selects conjectures that discharge the failed POs, whilst minimising the number of new failed POs that are introduced. This iterative approach to discovering all the missing invariants is typical of Event-B developments, as described in Section 5 of [BY08], where invariant discovery is manual. Of course, if

Failed PO:
p1 ↦ IDLEF ∈ statusF
t ∈ startFromM
p1 = from(t)
Fseqno(t) = currentSeqNo(p1)
⊢
p1 ∈ idleFP

Figure 9: Failed guard strengthening (GRD) PO resulting from a missing gluing invariant

a development is incorrect, then this process will not terminate. We return to the issue of working with incorrect developments in §7.

Note that the selection conjectures must be applied in order from SH1 to SH5 so as to optimise the selection procedure. Note as well that sometimes when a proof obligation fails it could mean that the prover could not handle it. As a way of coping with this, the invariant discovery process would terminate if none of the selected invariants help discharge a failed PO.

4.2 Mondex invariant discovery revisited

We now illustrate the application of our heuristics by returning to the refinement step described in §3.1. Recall that the gluing invariant (1) was required in order for the correctness of the refinement to be proved. When this invariant is missing from the model, an unprovable guard strengthening (*GRD*) PO[‡], as shown in Figure 9, is generated. The failed PO shows that the guard $p1 \in idleFP$ of the abstract event is not implied by the guards of the concrete event.

We start the process of invariant discovery with the application of heuristic CH1. We extract the list of core concepts that occur in the failed PO, giving them higher priority within the theory formation process. The extracted concepts are:

$$\{idleFP, statusF, startFromM, from, FSeqno, currentSeqNo\}$$

All of these concepts explicitly occur within the PO.

Regarding non-core concepts, the hypothesis $p1 \mapsto IDLEF \in statusF$ in the failed PO suggests that function *statusF* maps an arbitrary purse to the status *IDLEF*. This hypothesis rises the production of a non-core concept,namely the concept of all purses that map the status *IDLEF* in function *statusF*. This concept is obtained through the application of the split production rule over the concept *statusF* on the value *IDLEF*. No other non-core concepts are identified in the PO.

The next step is the selection of the production rules. The following production rules are used in the invariant discovery process:

$$\{compose, disjunct, negate, split\}$$

The compose, disjunct and negate production rules are always used in the search, as defined by heuristic CH2. The split production rule is selected when any reference to a member of a finite set occurs; in this context this reference is to constant IDLEF of set *status*. Thus, the split production rule is applied over the finite set *status* and the values to split are

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

all the members of the set, i.e.: *IDLEF, EPR, EPA, ABORTEPR, ABORTEPA, ENDF, IDLET, EPV, ABORTEPV* and *ENDT*.

After the application of the CH heuristics, the initial configuration of HR is complete. By running HR for 1000 steps, 2134 conjectures were formed. This should be compared with the 4545 conjectures that are generated if our CH heuristics are not used to configure HR.

Now turning to the SH heuristics, SH1 selects conjectures that relate to the prioritised concepts that appear within the goal of the failed PO. In our example, this focuses on conjectures that involve the concept *idleFP*. After applying SH1 we obtained:

4 equivalences, 2 implications and 79 non-exists conjectures

The application of SH2 removes conjectures whose left- and right-hand sides are not disjoint with respect to the variable occurrences. The application of SH2 yields the following results:

1 equivalence, 2 implications and 79 non-exists conjectures

Through the application of SH3, less general conjectures are removed. Applying this heuristic produces:

1 equivalence, 2 implications and 46 non-exists conjectures

SH4 selects only conjectures that discharge the failed PO, the results of this step are:

1 equivalence, 0 implications and 0 non-exists conjectures

Only one conjecture discharges the failed PO. Furthermore, this conjecture does not introduce any additional failures; therefore, it represents an invariant. Within HR the invariant takes the form:

$$\forall A, B.(state(A) \wedge purseSet(B) \wedge idleFP(A, B) \Leftrightarrow status(IDLEF) \wedge statusF(A, B, IDLEF))$$

which translates into the missing gluing invariant (1). It should be noted that this conjecture was formed by HR after one theory formation step. This shows that, in this example, our heuristics guided HR to discover interesting conjectures early within the theory formation process.

5 Invariant Discovery Implementation

In this section we describe a prototype implementation of our heuristic approach to invariant discovery. The architecture of the prototype is given in Figure 10. 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 – what we refer to as HREMO. Both the HR tool and the Rodin tool-set are implemented in Java; therefore, the implementation effort was carried on in the Java language. Below we describe each of the major components of our prototype.

5.1 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 providing domain information from an Event-B model, i.e. the animation traces and failed POs, to HREMO.

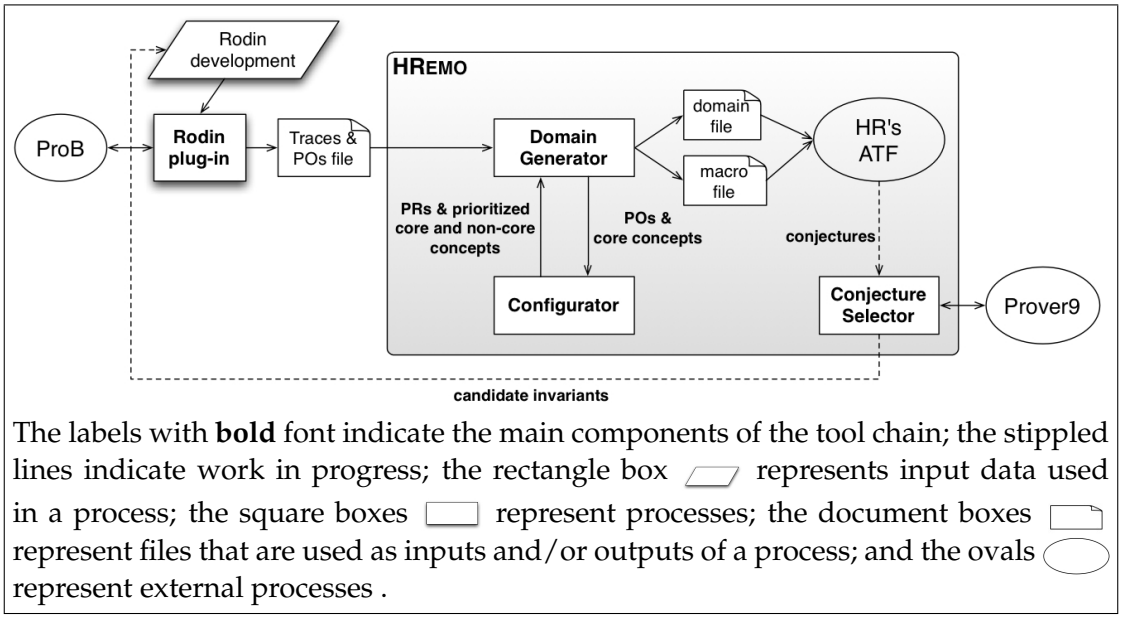

Figure 10: Tool-chain architecture

The plug-in receives as input a Rodin development from which the user has to select an Event-B machine in the refinement chain. The plug-in then extracts the information about the animation traces and failed POs from the selected machine and its abstract machine(s), i.e. the invariant discovery process is only applied to one step of the refinement at a time. This information is then output into a file. As it was mentioned previously, our long term aim is to support invariant discovery across a range of formalisms. To this end, the output of the plug-in consists of an xml file that must comply with a specific format imposed through a DTD schema. This schema specifies the structure of the information in the xml file.

As shown in Figure 10, the ProB animator is used for the simulation of Event-B models. Currently the Rodin tool-set does not support a test case generator, we relied instead on the random animation capability of ProB for the generation of traces. However, the quality of the discovered invariants depends on the quality of the traces obtained from the simulation of the models; therefore, randomly generated traces have the drawback that they may not contain a representative test suite, leading to the discovery of non-significant invariants. We believe the development of a test generator is necessary; however, this is currently outside of our scope and is left as future work. In the meantime, the plug-in gives the option to 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.

After the simulation of the Event-B model has finished, the plug-in produces the xml file that contains:

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

This file represents the input to HREMO.

5.2 Domain Generator

The main functionality of the *domain generator* is to process the traces and POs file 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 production rules as well as other parameters for the theory construction.

The traces contained in the xml file are transformed by converting each element of a model, i.e. variables, constants and sets, into Java objects that represent background concepts for HR. For each of these elements, their type is converted into parameters of the concept and the animation values into its examples, i.e. the concepts data tables, as illustrated for the Mondex system in §3. The result of this process is the core background information of the model as required by HR. The POs, on the other hand, are parsed into a tree-like structure that allows their analysis by the configurator; that is, the domain generator uses the configurator for the analysis of the failed POs in the configuration heuristics. The output of the configurator determines the order of the concepts in the domain file, the non-core concepts that are forced into the domain and the production rules that are specified in the macro file.

5.3 Configurator

The *configurator* receives the parsed POs and the core background concepts from the domain generator and apply the configuration heuristics in order to make the selection of the prioritised core and non-core concepts as well as the production rules from the failed POs.

The analysis of prioritised core and non-core concepts is as follows:

1. The configurator analyses the goals and then the hypotheses within the failed POs. This is because as mentioned in §4, 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. Goal and hypothesis formula are classified 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 $>, <, =, \in, \subseteq$, etc, while examples of unary operators are $\exists, \forall, \neg$, etc.
3. Through a recursive method, 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 and the identifier is equal to the name of a variable or constant in the domain background concepts.
   (b) A non-core concept is an aggregate of variables and constants that can be replicated in HR through the use of the production rules. In order to identify non-

```
method FIND_CORE_AND_NONCORE_CONCEPTS(formula)
  if isLiteral(formula) then
    literal = formula.identifier
    if literal ∈ domainVariables || literal ∈ domainConstants then
      addPrioritizedCoreConcept(formula)
      return TRUE
  else if isUnary(formula) then
    operator = formula.operator
    validFormula = FIND_CORE_AND_NONCORE_CONCEPTS(formula.uniFormula)
    if validFormula && isOpCompatible(operator) then
      addPrioritizedNonCoreConcept(formula)
      return TRUE
  else if isBinary(formula) then
    operator = formula.operator
    validLeft = FIND_CORE_AND_NONCORE_CONCEPTS(formula.left)
    validRight = FIND_CORE_AND_NONCORE_CONCEPTS(formula.right)
    if validLeft && validRight && isOpCompatible(operator) then
      addPrioritizedNonCoreConcept(formula)
      return TRUE
    else
      Map<Value, Type> splitValues = emptyMap
      List<Concept> coreConcepts = emptyList
      CHECK_SPLIT_CONCEPTS(formula, splitValues, coreConcepts)
      if splitValues.size > 0 && coreConcepts.size > 0 then
        foreach splitVal : splitValues.getValues do
          type = splitValues.getType(splitVal)
            foreach cc : coreConcepts do
              parametersTypes = cc.getParametersTypes
              if parametersTypes.contains(type) then
                BinaryFormula bf = new BinaryFormula(SPLIT, cc, splitVal)
                addNonCoreConcept(bf)
  return FALSE
end

  method CHECK_SPLIT_CONCEPTS(formula, splitValues, coreConcepts)
    if isLiteral(formula) then
      literal = formula.identifier
      if literal ∈ domainVariables || literal ∈ domainConstants then
        coreConcepts.add(domainConcepts.get(literal))
      else
        foreach set : domainSets do
          if set.getMembers.contains(literal) then
            splitValues.add(literal, set.getName)
    else if isUnary(formula) then
      CHECK_SPLIT_CONCEPTS(formula.uniFormula, splitValues, coreConcepts)
    else if isBinary(formula) then
      CHECK_SPLIT_CONCEPTS(formula.left, splitValues, coreConcepts)
      CHECK_SPLIT_CONCEPTS(formula.right, splitValues, coreConcepts)
    end end
```

Figure 11: Pseudo-code for the identification of core and non-core concepts. The method checks the type of the formula, i.e. whether it is literal, unary or binary. If the formula is a literal and the identifier of the formula is a variable or a constant, then a core concept has been found. For unary and binary formulas, if the operator is compatible with a production rule and the sub-formulas are themselves valid, then a non-core concept has been found. In addition, if the operator or the sub-formulas of a binary formula are not valid, it is checked if a non-core concept can be derived from the split production rule; that is, if the formula contains valid split values, i.e. members of a domain set, and a corresponding core concept for which the type of one of its parameters is equal to the type of the split value.

core concepts from a formula we analyse the compatibilities between formula operators and production rules. For instance, the conjunction and intersection operators ($\wedge$ and $\cap$) are compatible with the *compose* production rule. A binary or unary formula is a valid non-core concept if its operator is compatible with a production rule and its sub-formulas are themselves valid formulas.

(c) Moreover, a non-core concept can be derived from the application of the split production rule over a binary formula if:
- the formula is not valid, i.e. the binary operator is not compatible with a production rule or one or both sub-formulas are not valid;
- the formula contains valid split values; i.e. valid split values are those that are members of a domain set, e.g. the value *green* is a member of the domain set *Color*; therefore, *green* is a valid split value; and
- the formula contains a core concept that is compatible with the split value; i.e. a core concept from which one of its parameters could be assigned the split value, e.g. the core concept *ml_tl*, which denotes a traffic light, could be assigned the value *green*.

The pseudo-code for the identification of core and non-core concepts is given in Figure 11. The same method is used to identify the production rules that are used during the conjecture search in the theory formation process. That is, when an operator is found that is compatible with a production rule, that production rule is enabled for the search.

5.4 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. This is achieved by applying the selection heuristics as follows:

1. Select implication and equivalence conjectures for which either the left- or the right-hand side of the conjecture is a prioritised core or non-core concept, while non-existence conjectures are selected if a prioritised core or non-core concept occurs within the conjecture.
2. The concepts of each side of the implication and equivalence conjectures are compared and if the same concept appears in both sides, the conjecture is removed.
3. Prover9 [McC10] is used to choose the most general conjectures. Prover9 is an automated theorem prover for first-order and equational logic and is the successor of Otter [McC03], the theorem prover 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 (SH4) and that produce less extra failed POs (SH5) are the next steps of the process. We have implemented heuristics SH1, SH2 and SH3. Heuristics SH4 and SH5 are currently performed manually since they are language- and tool-dependent; for instance, the generation of POs in a formal model requires specialised mechanisms that are particular to each formalism. As

part of future work we would look at ways of giving a more generic solution to the application of these heuristics.

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

6 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 1 and 2 of Abrial's cars on a bridge system [Abr10]:* 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.

We used the second stage of our experiments to *evaluate* the heuristics developed during stage one. Here the experiments were performed on more complex Event-B models:

1. *Refinement 3 of Abrial's cars on a bridge system [Abr10]:* The third refinement of this system models the introduction of sensors that detect the physical presence of cars.
2. *The Mondex system [BY08]:* 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 targeted the third, fourth and eighth refinement steps. The third refinement handles dual state sets in both sides of a transaction in order to handle information locally. The fourth refinement introduces the use of messaging channels between purses and the eighth refinement introduces a status function that maps purses to their states instead of using state sets.

In the work reported in [BY08], 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 attempted the re-construction of the Mondex system in Event-B based on the

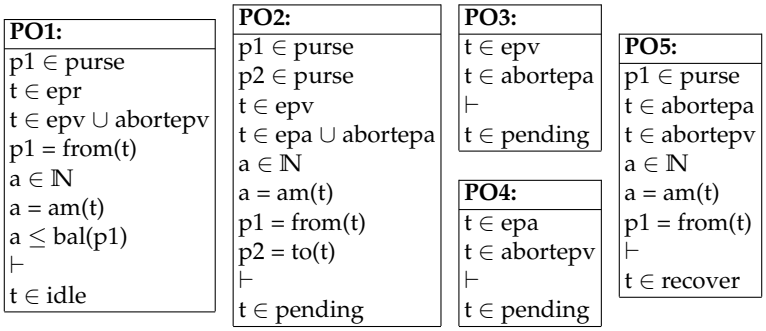

Figure 12: First set of failed POs.

development presented in [BY08]. In the following section we present the results obtained by the application of our approach to the refinement between levels three and four of the Mondex system, and we show that these results are similar to the ones obtained through the interactive development [BY08].

6.1 The Mondex system

In level three of the Mondex system a transaction is permitted to be in one of four states: *idle, pending, recover or ended*, while the refinement in level four introduces dual states to a transaction so that each side has their own local protocol state. In order to evaluate our approach, we introduced the model in level 4 with only basic typing invariants. The absence of the invariants produces the failed POs shown in Figure 12.

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

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

Moreover, from the analysis of the predicates in the failed POs, we identify the following non-core concepts:

$$\{epv \cup abortepv, epa \cup abortepa\}$$

These concepts are identified from hypotheses $t \in epv \cup abortepv$ and $t \in epa \cup abortepa$ within PO1 and PO2, respectively. Note that t does not represent a concept in the domain, it represents an arbitrary transaction passed as a parameter to the event associated with the failed POs. For this reason, only the right hand sides of the membership relations are selected as interesting non-core concepts.

The process continues with the selection of the productions rules. Based on the failed POs shown in Figure 12, the following production rules are selected for the search:

$$\{compose, disjunct, negate, numrelation\}$$

The compose, disjunct and negate production rules are always used in the search as stated in heuristic CH2. The numrelation production rule is selected because hypothesis $a \leq bal(p1)$ within PO1 expresses a property based on the relational operator $\leq$. After the pre-heuristics have been applied HR is run for 1000 steps, resulting in 7296 conjectures.

The selection heuristics are now applied over this set of conjectures. Heuristic SH1 suggests looking at the prioritised concepts associated to the goals of the failed POs. From

Heuristic	Concept	Equivalences	Implications	Non-exists
SH1	idle	7	27	24
	pending	6	27	35
	recover	9	51	41
SH2	idle	0	27	24
	pending	0	27	35
	recover	2	51	41
SH3	idle	0	6	17
	pending	0	8	26
	recover	2	3	30
SH4	idle	0	2	0
	pending	0	2	0
	recover	1	0	0

Table 1: Results of the application of selection heuristics SH1, SH2, SH3 and SH4.

the goals of the POs shown in Figure 12, we identified the concepts *idle*, *pending* and *recover*. Thus, we look for the conjectures associated to each of these concepts. The results from the application of this heuristic are shown in Table 6.1. This table also shows the results of applying heuristics SH2, SH3 and SH4 over each of the selected concepts.

As can be observed, after applying the four initial selection heuristics we have narrowed the set of selected conjectures to a total of five conjectures: two implications involving the concept *idle*, two implications for concept *pending* and one equivalence about the concept *recover*.

The final step in the discovery process is the selection of the conjectures that produce the smaller number of new failed POs. The two implications associated with concept *idle* discharge PO1 and produce one extra failed PO. We believe that in this kind of situation it is the user who has to decide which one is the most appropriate conjecture according to his/her knowledge about the model. Thus, we present both conjectures as candidate invariants and leave the decision of which one to select to the user. Regarding the two implications 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 to 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. The set of invariants represented by the conjectures obtained from HR in this first iteration of our approach is shown in the results Table 4 in the row of level 3-4 of the Mondex case study[§].

After the new set of invariants is introduced to the model, six new failed POs are generated. We then start the analysis again by applying our approach based on the new set of failed POs. This new iteration results in the discovery of five new invariants. Again, when these invariants are added to the model, one new failed PO is generated. We discovered one

[§]Note that we have given the equivalent set theoretic representation of these 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 [BY08], have shown that the automatic provers do better with quantifier-free predicates.

Event-B model	Step	Failed POs	Invariants				
			Automatically discovered				
			Glue	System	Total	Iteration	
Traffic light	Level 1-2	2	2	0	**2**	1	Development set
Vending machine (Arith)	Level 1-2	6	3	0	**3**	1	
Vending machine (sets)	Level 1-2	6	3	0	**3**	1	
Cars on a bridge	Level 1-2	2	1	0	**1**	1	
	Level 2-3	6	0	5	**5**	1	
	Level 3-4	7	0	5	**5**	1	
Mondex	Level 3-4	5	5	0	**5**	1	Evaluation set
		6	1	4	**5**	2	
		1	0	1	**1**	3	
	Level 4-5	3	0	3	**3**	1	
		5	0	4	**4**	2	
		4	0	2	**2**	3	
	Level 8-9	14	10	0	**10**	1	

Table 2: Automatically discovered invariants.

new invariant after a third iteration of our approach. No further failed POs are generated when this invariant is added to the model. Table 4 shows the invariants obtained after the second and the third iteration, respectively.

The invariants shown in Table 4 for level 3-4 of the Mondex case study are a subset of the invariants suggested in [BY08] for this step of the refinement. In total we obtained 11 invariants from the 17 used in [BY08]. However, it is important to note that we have addressed all the failures produced when proving consistency between the refinement levels. Our hypothesis is that the extra invariants used in [BY08] represent new requirements of the system, which are out of the scope of our technique since we only target invariants needed to prove the refinement steps.

6.2 Summary of results

Table 2 summarises the results of the application of our approach in each of the Event-B models used during the development and the evaluation stages. Notice that all the experiments were performed over models with only basic typing invariants. This means that neither gluing nor system invariants were present in the models when using our technique. The table shows for each refinement step, the number of failed POs that arose, 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.

In Table 3 we compare our results with the actual invariants given in the literature for the models of the cars on a bridge [Abr10] and the Mondex system [BY08]; the other developments are not compared because they are of our own authorship (note that the invariants of the refinement from levels four to five of the Mondex system are not given in the literature). All automatically discovered invariants are subsets of the invariants given in the literature; however, it is important to highlight that the automatically discovered invariants

Event-B model	Step	Given in Literature			Automatically discovered		
		Glue	System	Total	Glue	System	Total
Cars on a bridge	Level 1-2	1	1	**2**	1	0	**1**
	Level 2-3	0	5	**5**	0	5	**5**
	Level 3-4	0	23	**23**	0	5	**5**
Mondex	Level 3-4	8	9	**17**	6	5	**11**
	Level 4-5	-	-	-	0	9	**9**
	Level 8-9	10	0	**10**	10	0	**10**

Table 3: Comparison between hand-crafted and automatically discovered invariants.

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.

were sufficient to prove all the refinement steps in our experimental models.

As it can be observed from Table 3, the automatic discovery of gluing invariants through the use of theory formation and the HR system has provided promising results. In most cases, the set of gluing invariants discovered through our technique was almost identical to the set of gluing invariants provided in the literature. Regarding 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 our approach. As mentioned previously, we believe that this difference can be explained by the introduction of new requirements, resulting in the need for extra properties in the model.

Table 4 shows all the invariants that were discovered through the application of our approach.

7 Related and future work

As far as we are aware, automated theory formation techniques have not been investigated within the context of refinement style formal modelling. The closest work we know of is Daikon [EPG+07], a system which uses templates to detect "likely" program invariants by analysing program execution traces. In the following section we present a comparative study of our approach with Daikon.

7.1 Comparative study with the Daikon system

Daikon [EPG+07] observes the values computed in a program execution and derives properties that are true over such execution. Daikon detects invariants for different programming languages, i.e. 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, Daikon evaluates selected variables at specified points within the code, these are called *program points*. Program points are usually object and procedures entries and exits, and variables are either: *program variables*, which are those explicitly written in the program code, e.g. class instance variables, procedure parameters, return values,

Case study	Step	Automatically discovered Invariants		
		Iteration 1	**Iteration 2**	**Iteration 3**
Traffic Light	Level 1-2	r_light=TRUE $\vee$ a_light=TRUE $\Leftrightarrow$ red_light=TRUE sold = soldMilk + soldPlain		
Vending machine (arith rep.)	Level 1-2	stock = stockMilk + stockPlain g_light = TRUE $\Leftrightarrow$ green_light = TRUE givenCoin = EMPTY_COIN $\Leftrightarrow$ coin = NO_COIN		
Vending machine (sets rep.)	Level 1-2	available = productStatus^{-1}[{AVAILABLE}] limited = productStatus^{-1}[{LIMITED}] soldOut = productStatus^{-1}[{SOLDOUT}]		
Cars on a bridge	Level 1-2	n = a + b + c		
	Level 2-3	ml_tl = green $\Rightarrow$ c = 0 il_tl = green $\Rightarrow$ a = 0 ml_tl = red $\Rightarrow$ ml_pass = 1 il_tl = red $\Rightarrow$ il_pass = 1 il_tl = green $\Rightarrow$ ml_tl = red		
	Level 3-4	ml_out_10 = TRUE $\Rightarrow$ ml_tl = green il_out_10 = TRUE $\Rightarrow$ il_tl = green IL_IN_SR = on $\Rightarrow$ A $>$ 0 IL_OUT_SR = on $\Rightarrow$ B $>$ 0 ML_IN_SR = on $\Rightarrow$ C $>$ 0		
Mondex	Level 3-4	epr $\subseteq$ idle (idleF $\cup$ epr) $\subseteq$ idle epv $\cap$ (epa $\cup$ abortepa) $\subseteq$ pending epa $\cap$ (epv $\cup$ abortepv) $\subseteq$ pending abortepa $\cap$ abortepv = recover	idleF $\subseteq$ idle idleT $\cap$ (epa $\cup$ abortepa) $= \varnothing$ idleT $\cap$ (epv $\cup$ abortepv) $= \varnothing$ epv $\cap$ abortepv $= \varnothing$ epa $\cap$ abortepa $= \varnothing$	epr $\cap$ idleF $= \varnothing$
	Level 4-5	epr $\cap$ reqM $\subseteq$ epv $\cup$ abortepv epv $\cap$ valM $\subseteq$ epa $\cup$ abortepa endT = endF $\cup$ ackM	reqM $\cap$ idleF $\subseteq$ epv $\cup$ abortepv valM $\cap$ idleT $= \varnothing$ epr $\cap$ valM $= \varnothing$ epr $\cap$ abortepa $= \varnothing$	valM $\cap$ idleF $= \varnothing$ abortepa $\cap$ idleF $= \varnothing$
	Level 8-9	idleFP = statusF^{-1}[{IDLEF}] eprP = statusF^{-1}[{EPR}] epaP = statusF^{-1}[{EPA}] aborteprP = statusF^{-1}[{ABORTEPR}] abortepaP = statusF^{-1}[{ABORTEPA}] endFP = statusF^{-1}[{ENDF}] idleTP = statusF^{-1}[{IDLET}] epvP = statusF^{-1}[{EPV}] abortepvP = statusF^{-1}[{ABORTEPV}] endTP = statusF^{-1}[{ENDT}]		

Table 4: Automatically discovered invariants.

etc.; 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 templates[¶], which are:

- instantiated using the set of selected variables;
- evaluated against the execution values; and
- removed if the instantiated template does not hold for all the states of the execution.

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.

[¶]In [EPG+07] it was reported that Daikon contained 75 invariant templates and 25 derived variable templates.

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 apply 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 are 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.

The approaches however 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 produces invariants that represent pre- and/or post-conditions of specific procedures within the code, while HREMO only generates invariants that hold before and after the execution of each procedure, i.e. event, 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 generation tool for refinement based formal methods.

Experiments

We carried out two experiments 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 [Abr10] 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 in 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 with ProB.

Experiment 1: First refinement. As mentioned above, "cars on a bridge" 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 13 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 13 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.

Table 5 shows the results of the invariant analysis from both approaches for the models shown in Figure 13. 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. 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 to the absence of the invariant HREMO fails to identify it 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 it can be observed in Table 5, the outputs from Daikon for the entry and exit points of the method *ml_out* in the two Java implementations are different from each other. 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. We do not present the Event-B and Java developments here; however, details of the Event-B development can be found in [Abr10]. In terms of translating Event-B into Java, we followed the same approach as described for experiment 1. The results from this exper-

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

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

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

Table 5: Comparison of expected and detected invariants for the first refinement of the "Cars on a bridge" model.

iment are shown in Table 6. 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 we can again observe 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 color 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* $\Rightarrow$ *c=0*, which states that whenever the mainland traffic light is green, there are no cars travelling in the opposite direction.

Daikon provides a rather 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

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

Table 6: Comparison of expected and detected invariants for the second refinement of the "Cars on a bridge" model.

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, etc. Because of the vast number of available options it is difficult to find the optimal settings in which to run Daikon. We ran Daikon 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. We tried selecting setting options that could help influence the behaviour of Daikon, for instance enabling a particular invariant template, but we could not identify any useful option for our models apart from the templates that were enabled by default. Furthermore, reading the Daikon documentation we realised 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. Like Daikon, HREMO is configurable; however, 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 we could conclude 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 to 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 by 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. On the contrary, HR is a general purpose tool which is not restricted by patterns of conjectures and, moreover, the iterative application of the production rules provides greater flexibility in terms of the kinds of invariants that can be discovered.

7.2 Using other ATF systems for automated invariant discovery

We believe that the generation of invariants is not necessarily dependent on a specific ATF system (HR). Within automated theory formation there are a number of alternative tools to HR that could be explored. For instance, the IsaScheme system [MRMDB11] implements a scheme-based approach to ATF. Schemes are higher-order formulae which can be used to generate new concepts and conjectures; variables within the scheme are instantiated automatically and this drives the invention process. Montaño-Rivas has enabled IsaScheme to automatically discover invariants by handcrafting 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 which we have found using the HR system. The biggest disadvantage at present is that the schemata need to be fine-tuned to match specific invariants; such fine-tuning is only justified if they can be used to invent further invariants. Even with the addition of further schemata, the schema-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 schemata there are, the more time the system would take to generate all invariants, so further schemata may detract from the efficiency of IsaScheme.

Other examples of AFT and MTE (mathematical theory exploration) systems which might find application in this domain include IsaCoSy [JDB10], the CORE system [MIDA09] and MATHsAiD [MBA07]. Underlying the first two of these systems is a notion of *term synthesis*, i.e. the automatic generation of candidate conjectures based upon application of domain knowledge. Like IsaScheme, IsaCoSy supports the discovery of theorems within the context of mathematical induction, while MATHsAiD provides broader support for the development of mathematical theories. The CORE system has a strong software verification

focus, supporting the automatic generation of frame and loop invariants for use in reasoning about pointer programs.

7.3 Future work

As noted above, animation is a key aspect of our 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 our experiments; however, we found two areas where further improvements are required:

1. We believe that increasing the randomness in the production of the traces would improve our results.
2. ProB preferences only allow for the creation of sets with a few elements, as well as very limited integer ranges. This restricted some of the traces we were able to generate, and thus impacted negatively on the invariants that could be discovered. Specifically, this limitation arose during our analysis of the Mondex case study.

The process of finding a "correct" refinement will typically involve exploring many "incorrect" refinements. While the work reported here focuses on supporting the verification of correct refinements, we are currently investigating how counter-examples generated by ProB could be combined with HREMO in order to provide useful feedback to a developer when faced with an incorrect refinement.

In order to show that ATF techniques in general can be used for automated invariant discovery, we hope to explore the use of other ATF systems in this context. In particular, we plan to collaborate with Montaño-Rivas to further develop this application in his IsaScheme system [MRMDB11].

Longer-term, we are aiming at using theory formation within our REMO [IGLB11] formal modelling planning system. That is, when faced with a refinement failure, we aim to use theory formation, automatically tailored by refinement plans [LGI10], to suggest modelling guidance. Of course, such "modelling guidance" are only suggestions, ultimately users must select which is most appropriate for their needs.

Currently most of the invariant discovery process within HREMO has been automated; however, heuristics SH4 and SH5 are still to be implemented. As mentioned above, in order to automate these heuristics we require tool capabilities which are dependent of each formalism, e.g. a proof obligation generator. As future work, we aim to automate this process and integrate it with our REMO tool.

8 Conclusions

Building upon HR, animation and proof-failure analysis, we have presented HREMO– an automatic approach to invariant discovery. The key contribution of our work are the heuristics we have developed. Using proof-failure analysis to prune the wealth of conjectures HR discovers, these heuristics have proven highly effective at identifying missing invariants. While more experimentation is required, we believe that our heuristics 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 developers.

Acknowledgements: Our thanks go to Alan Bundy, Gudmund Grov and Julian Gutierrez for their feedback and encouragement with this work. Also, we want to thank Jens Bendisposto and the ProB development team for their assistance, Simon Colton and John Charnley for their help in using the HR system, and Omar Montaño-Rivas for his thoughts on how IsaScheme could be applied to automated invariant generation. The research reported in this paper is supported by EPSRC grants EP/F037058, EP/F035594 and EP/J001058.

References

[ABH+10] J-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[Abr10] J-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[BBCJ+05] B. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 2005.

[BY08] M. Butler and D. Yadav. An incremental development of the Mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.

[CBW00] S. Colton, A. Bundy, and T. Walsh. Automatic invention of integer sequences. In *16th IJCAI*, pages 786–791, 2000.

[CM01] S. Colton and I. Miguel. Constraint generation via automated theory formation. In *7th International Conference on the Principles and Practice of Constraint Programming*, 2001.

[Col02] S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag, 2002.

[CP05] S Colton and A Pease. The TM system for repairing non-theorems. In *Selected papers from the IJCAR'04 disproving workshop, Electronic Notes in Theoretical Computer Science*, volume 125(3). Elsevier, 2005.

[EPG+07] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[Eps88] S. L. Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4(1):42–53, 1988.

[Faj88] S. Fajtlowicz. On conjectures of Graffiti. *Discrete Mathematics*, 72:113–118, 1988.

[Had49] J. Hadamard. *The Psychology of Invention in the Mathematical Field*. Dover, 1949.

[IGLB11] A. Ireland, G. Grov, M. Llano, and M. Butler. Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance. *Science of Computer Programming*, 2011.

[JDB10] M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *1st International Conference on Interactive Theorem Proving*, volume

6127 of *LNCS*, pages 291–306. Springer, 2010.

[Kop75] E. Koppelman. Progress in mathematics. In *Workshop on the Historical Development of Modern Mathematics*, July, 1975.

[Lak76] I. Lakatos. *Proofs and Refutations*. CUP, Cambridge, UK, 1976.

[Lam72] W. E. Lamon. *Learning and the Nature of Mathematics*. Science Research Associates, Palo Alto, 1972.

[LB03] M. Leuschel and M. Butler. Prob: A model checker for b. In *International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[Lef72] G. R. Lefrancols. *Psychological Theories and Human Learning*. Wadsworth Publishing, Belmont, Ca., 1972.

[Len76] D. Lenat. *AM: An Artificial Intelligence approach to discovery in mathematics*. PhD thesis, Stanford University, 1976.

[Len83] D. Lenat. Eurisko: A program which learns new heuristics and domain concepts. *Artificial Intelligence*, 21, 1983.

[LGI10] M. Llano, G. Grov, and A. Ireland. Automatic guidance for refinement based formal methods. *5th workshop on Automated Formal Methods (AFM'10), a satellite workshop of the 22nd International Conference on Computer Aided Verification (CAV'10).*, 2010. Also available via: School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0076; School of Informatics, University of Edinburgh, Report EDI-INF-RR-1371.

[MBA07] 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) of *Studies in Logic, Grammar and Rhetoric*, pages 135–149. University of Białystok, 2007.

[MBS06] Roy McCasland, Alan Bundy, and Patrick Smith. Ascertaining mathematical theorems. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 151(1), pages 21–38. Elsevier, 2006.

[McC03] W. McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003.

[McC10] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[MIDA09] E. Maclean, A. Ireland, L. Dixon, and R. Atkey. Refinement and term synthesis in loop invariant generation. In *2nd International Workshop on Invariant Generation (WING'09), a satellite workshop of ETAPS'09*, 2009.

[MRMB11] O. Montaño-Rivas, L. McCasland, R. Dixon, and A. Bundy. Scheme-based theorem discovery and concept invention. In *Expert Systems with Applications*, 2011.

[MRMDB11] Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, (0):–, 2011.

[MSC02] A. Meier, V. Sorge, and S. Colton. Employing theory formation to guide proof planning. In *AISC/Calculemus'02, LNAI 2385*. Springer, 2002.

[Pea07] A. Pease. *A Computational Model of Lakatos-style Reasoning*. PhD thesis, School of Informatics, University of Edinburgh, 2007. Online at

http://hdl.handle.net/1842/2113.

[Pol45] G. Polya. *How to solve it*. Princeton University Press, 1945.

[Pol54a] G. Polya. *Mathematics and plausible reasoning: Induction and analogy in mathematics*, volume I. Princeton University Press, 1954.

[Pol54b] G. Polya. *Mathematics and plausible reasoning: Patterns of Plausible Inference*, volume II. Princeton University Press, 1954.

[Pol62] G. Polya. *Mathematical Discovery*. John Wiley and Sons, New York, 1962.

[PSC+10] 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, PA, USA, 2010.

[RH90] G. Ritchie and F. Hanna. AM: a case study in methodology. In D. Partridge and Y. Wilks, editors, *The foundations of AI: a sourcebook*, pages 247–265. CUP, Cambridge, 1990.

[SB89] M.H. Sims and J.L. Bresina. Discovering mathematical operator definitions. In *Proceedings of the Sixth International Workshop on Machine Learning*, S.F., CA., 1989. Morgan Kaufmann.

[SB06] 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.