Discovery of invariants through automated theory formation

Under consideration for publication in Formal Aspects of Computing

Discovery of Invariants through Automated Theory Formation

Maria Teresa Llano[1] and Andrew Ireland[1] and Alison Pease[2,3]

[1]School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK
[2]Department of Computing
Imperial College London, UK
[3]Department of Computer Science
Queen Mary, University of London, 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. 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: Automated invariant discovery; Automated theory formation; Formal modelling and refinement; heuristic approaches to invariant discovery; Event-B; HR

1. Introduction

By allowing a developer to introduce design steps incrementally, 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 which relate to their design decisions. Ideally,

Correspondence and offprint requests to: Maria Teresa Llano, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, UK. e-mail: mtllano@gmail.com

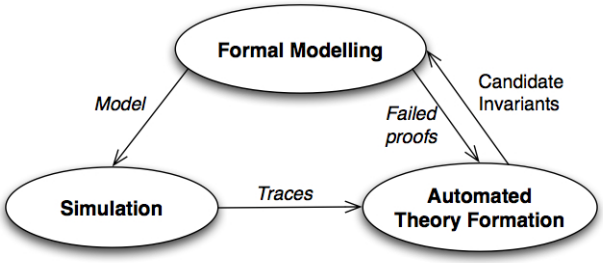


Fig. 1. Approach for the automatic discovery of invariants.

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;
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 tool-set [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, HR [Col02a]. 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, our focus is on formal models. We discuss this further in Section 8.

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 challenges: 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 have 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 and our implementation relies on the specificity of this representation, we believe that our approach is sufficiently general to be applied to any refinement style formal modelling framework which supports simulation and uses proof in order to verify refinement steps.

1.1. Motivation and context

Refinement is a technique used to construct formal models. It provides a way of handling the complexity of modelling large systems by adding detailed functionality progressively through incremental steps. Refinement starts with the introduction of an abstract model which captures the main functionality of the system and is then refined by adding more concrete information about its behaviour.

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

introduction of errors when changes are made to a model; conversely, their absence increases the possibilities of errors being introduced into a model when the system evolves.

Proof-failure analysis is often used to discover invariants which are required to prove refinement steps. For instance, in the model of the Mondex electronic purse system developed in [BY08], the authors describe an iterative process in which they manually strengthened invariants in order to overcome refinement proof-failures. Additionally, in [Abr10] Abrial highlights that the productive use of proof-failure not only aids one's understanding of a model, but also assists in the discovery and strengthening of invariants. In particular, Abrial uses proof-failure to obtain clues about the invariants which are required to prove strong synchronisation between an action and a reaction within a mechanical press controller system [Abr10, Chap. 3]. Automating this analysis would increase the productivity of users and improve the accessibility of formal modelling methods.

A wide spectrum of approaches to automatic invariant discovery have been explored in the literature. Some are dynamic in nature, such as the Daikon system [EPG+07] which uses invariant templates to analyse program execution traces. Others are constraint-based, for example in [Bol05] the Alloy analyser is used to discover retrieve relations for Z specifications. High-level patterns of proof, in the form of proof plans, have also been shown to be effective in constraining the search for program invariants [IEC+06, MIG11]. In the work presented here we use a novel technique based on ATF to analyse simulation traces of a formal model in order to find the invariants required to prove a refinement step.

Our approach assumes that the refinement step being analysed is correct; that is, the concrete model is consistent with the abstract model. However, the process of finding a "correct" refinement typically involves exploring many incorrect models. In order to cope with the possible occurrence of an incorrect model, the invariant discovery process terminates when no failed PO is discharged by the set of proposed invariants. This also addresses cases in which the required invariant(s) cannot be generated through our technique – examples of this case will be given in Sections 7.1.1 and 7.2.

1.2. Contributions and organisation

The main contribution of our work is the development of a heuristic approach to the automatic discovery of invariants of formal models through ATF. We present a prototype system called HRemo, which implements our approach, and provide empirical evidence for its effectiveness, based upon several case studies from the literature. While our principal goal is to develop and explore techniques which support formal refinement, a secondary outcome of our research is to extend the field of Automated Theory Formation by employing and extending ATF techniques within a new domain. While these dual contributions cannot easily be separated, we can see them in the following way:

- Contributions to formal refinements. We:
  - present an automatic approach to invariant discovery;
  - demonstrate how techniques from ATF can be used to automatically generate invariants;
  - develop a set of heuristics which tailor the ATF routine to the invariant discovery process;
  - use proof-failure analysis to prune the set of conjectures which have been automatically produced;
  - present an implementation of a prototype system, HRemo, which embodies our ideas; and
  - present a set of experiments that illustrate the application of our technique.
- Contributions to ATF. We:
  - extend ATF by employing their techniques in a new domain;
  - demonstrate ideas using the leading system in the field, HR;
  - automate the construction of the domain and macro files, which are input to HR prior to a run; and
  - automate the selection of conjectures which are output by HR, after a run, to filter those of particular interest.

An earlier workshop version of this paper appeared in [LIP11]. Here we provide a significantly more detailed account of our methodology and technique for automating the discovery of invariants. We also report on a prototype implementation along with additional case study material. Initial findings of a comparative study

with the Daikon invariant generation system are included. An explanation is also provided as to how our approach could be applied to other formal methods, using the Z notation as an example.

The remainder of this paper is organised as follows. In Section 2 we provide background on both Event-B and HR. The application of HR within the context of Event-B is described in Section 3, along with the limitations highlighted above. In Section 4 we present our heuristics, and describe their rationale. We illustrate the application of our heuristics in Section 5. A description of the prototype implementation is presented in Section 6. Our experimental results are given in Section 7, while a comparison with Daikon, and other related techniques are discussed in Section 8. Our conclusions and future work plans are presented in Section 9.

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 *contexts* and *models*. 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. This case study models a protocol for money transfer that ensures that no money is lost in a transaction regardless of the success or failure of the transaction. The model developed in [BY08] is composed of one abstract model and eight refinement steps. The tasks performed at each level follow.

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

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

Level 3: removes some redundant variables from the model.

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

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

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

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

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

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

Details of each of these steps can be found in [BY08]. Here we focus on the last refinement step, which is shown in Figure 2, in order to illustrate the concept of refinement within Event-B.

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, when an abstract event is refined, the keyword **refines** indicates the refined event and occurs in the definition of 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

Abstract Level

Context:
Sets
purseSet

Model:
Variables
idleFP, eprP, epaP, aborteprP, abortepaP,
endFP, idleTP, epvP, abortepvP, endTP
Invariants
idleFP ⊆ purseSet
eprP ⊆ purseSet
epaP ⊆ purseSet
aborteprP ⊆ purseSet
abortepaP ⊆ purseSet
endFP ⊆ purseSet
idleTP ⊆ purseSet
epvP ⊆ purseSet
abortepvP ⊆ purseSet
endTP ⊆ purseSet
Events
...
StartFrom ≙
any t, p1
where
p1 ∈ idleFP
...
then
eprP := eprP ∪ {$p1$}
idleFP := idleFP \ {$p1$}
...
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 ∈ purseSet ⇸ status
Events
...
StartFrom ≙
refines StartFrom
any t, p1
where
p1 ↦ IDLEF ∈ 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.

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

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,[1] 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:

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

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.

[1] 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*).

2.2. Automated theory formation and HR

2.2.1. First steps in automated theory formation

Automated theory formation (ATF) derives its terminology from psychology in which the term "concept formation" is used (see, for example, [BGA67]) to describe the search for features which differentiate exemplars from non-exemplars of various categories. Lenat first used this term in his 1977 paper: *Automated Theory Formation in Mathematics* [Len77]. At this time, there were systems which could define new concepts for investigation, such as those described in [Win70], and systems which could discover relationships among known concepts, such as Meta-Dendral [Buc75], but no system could perform both of these tasks. Lenat saw this as the next step and built the AM system [Len77], designed to both construct new concepts and conjecture relationships between them, thus fully automating the cycle of discovery in mathematics. Lenat describes this as follows:

> What we are describing is a computer program which defines new concepts, investigates them, notices regularities in the data about them, and conjectures relationships between them. This new information is used by the program to evaluate the newly-defined concepts, concentrate upon the most interesting ones, and iterate the entire process. [Len77, p. 834]

AM is a rule-based system which uses a frame-like scheme to represent its knowledge, enlarges its knowledge base via its collection of heuristic rules, and controls the firing of these rules via its agenda mechanism. Lenat chose elementary arithmetic as the development domain because he could use personal introspection for the heuristics for constructing and evaluating concepts. Given the age of this discipline, Lenat thought it unlikely that AM would make significant discoveries, although he did cite its "ultimate achievements" as the concepts and conjectures it discovered (or could have discovered). He suggested various criteria by which his system could be evaluated, many of which focused on an exploration of the techniques. For instance, he considered generality (running AM in new domains) and how finely-tuned various aspects of the program are (the agenda, the interaction of the heuristics, etc).

Lenat saw his system and future developments in this field as having implications for mathematics itself (finding results of significance), for automating mathematics research (developing AI techniques), and for designing "scientist assistant" programs (aids for mathematicians). Despite the seeming success of the AM system, it is one of the most criticised pieces of AI research. In their case study in methodology, Ritchie and Hanna analysed Lenat's written work on AM and found that there was a large discrepancy between his theoretical claims and the implemented program [RH90]. For instance, Lenat made claims about how AM invented natural numbers from sets, whereas it used one heuristic which was specifically written in order to make this connection (and not used in any other context). Another problem was that the processes were sometimes under-explained. ([Col02a, Chap. 13] presents a more comprehensive history of AFT and a critique of Lenat's work.)

2.2.2. Contemporary approaches to automated theory formation

Colton and colleagues have developed a research programme which extends the term ATF. This consists of a family of techniques used to construct and evaluate concepts, conjectures, examples and proofs. The programme describes a model of mathematical theory formation whereby concepts are invented to describe and categorise examples of mathematical objects such as numbers, graphs or groups; conjectures are made which relate the concepts; and proofs and counterexamples are sought to determine the truth of the conjectures. There are varying motivations behind this programme. The primary motivation is the desire to discover powerful, generic algorithms for applications beyond domains of mathematics. A second motivation is to add to the body of mathematical knowledge, by software helping mathematicians to derive results that they otherwise would not find, and by software autonomously finding novel concepts, theorems and proofs. A third motivation is to study creative processes in people and in software. Simulating the creation of concepts, discovery of conjectures, derivation of proofs and finding of counterexamples, and by simulating ways in which all of these processes can interact, can shed light on ways in which human mathematicians create/discover new mathematics, and suggests more generic ways in which creative software can be built. All three motivations have played a part in the development of ATF. Colton's approach has led to the implementation of novel, generic AI algorithms, for machine learning, constraint solving, theorem proving, computer algebra and combinations thereof. His approach has also been used in mathematical discovery tasks, leading to new

results published in the literature. Moreover, ATF is one of the approaches studied within Computational Creativity research, and has added to our understanding of creative processes in people and software.

Each theory formation step consists in a single application of the following procedure:

Stage 1: New concepts are generated from old ones using one of a number of production rules, as described in [CBW99]. Concepts comprise a number of facets, most notably a formal definition in one of a number of representations (e.g., in first order logic when first order theorem provers are being used as part of the ATF routine), and a set of example tuples which satisfy the definition (these can be seen as the success set in logic programming terminology).

Stage 2: Conjectures relating concepts are formed empirically by looking for patterns in the examples satisfying the concepts, as described in [Col02b]. In particular, if no example tuples can be found for a concept, a non-existence conjecture is made stating that the concept's definition is inconsistent with the axioms of the domain. Similarly, if the tuples satisfying concept C all satisfy the definition of concept D, then an implication conjecture $C \rightarrow D$ is made, and if C and D have exactly the same examples, then the conjecture $C \leftrightarrow D$ is made.

Stage 3: Third party theorem provers such as Otter [McC94b] are employed to try and prove each conjecture, and if these fail, then model generators such as MACE [McC94a] are used to try and find a counterexample. If a proof is found for an equivalence conjecture $C \leftrightarrow D$, then concept D is not allowed into the theory, as doing so would result in a duplication of effort. Similarly if a non-existence conjecture is proved, then the concept with the inconsistent definition is not allowed into the theory. If a counterexample is found, then this usually results in the concept being allowed into the theory, and the counterexample being used to try and disprove other open conjectures.

Colton introduced this routine in [Col02a] and expanded upon it in [CM06]. Each application of the routine is an attempt to add a new concept to the theory: this attempt may result in a new concept, a conjecture (with or without a proof), a new example (introduced as a counterexample) or nothing being added to the theory. In this way, a rich theory can be built up. The theory formation is driven by a heuristic search which uses measures of interestingness to determine which old concept(s) to build new ones from at each application of the ATF routine. Such measures of interestingness are described in [CBW00c].

The value of the ATF approach has been demonstrated on numerous occasions, in the following ways:

- Through the discovery of new results published in the mathematical literature, e.g., in number theory [Col99] and – as part of a larger system comprising numerous automated reasoning programs – in loop and quasigroup theory [SCMM08, SMMC08].
- By contributing to previously human-only mathematical databases, such as the Encyclopedia of Integer Sequences [CBW00b] and the TPTP library [CS02].
- Through the improvement, enhancement and comparison of standard AI techniques such as constraint solving [CCM06], machine learning [CBW00a] and theorem proving [CP05, ZFCS02].
- By providing the inspiration and basis for more sophisticated mathematical theory formation approaches, based on philosophical or cognitive theories. For instance, the ATF routine is used in theory formation systems HRL [Pea07] and TM [CP04], which are based on Lakatos's [Lak76] characterisation of ways in which mathematicians respond to counterexamples and use them to evolve concepts, conjectures and proofs. Similarly, the routine forms the motivation for [Cha10, CC08], in which Charnley demonstrated a novel approach to combined reasoning by implementing a framework based upon the cognitive science theory of the Global Workspace [Baa88, Baa97].

For further details on the development of ATF and related fields, see [Col02a, CM06, PCCng].

2.2.3. The HR system

The ATF routine outlined above has been implemented firstly in Colton's HR1 system, a Prolog implementation described in [Col02a], and then in his HR2 system (known simply as HR[2]), a Java implementation described in [CM06]. HR is the leading system within the field of ATF, as evidenced by the number of publications which describe the system, and research projects which extend it (these are summarised in [CM06] and [PCCng], respectively). It 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

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

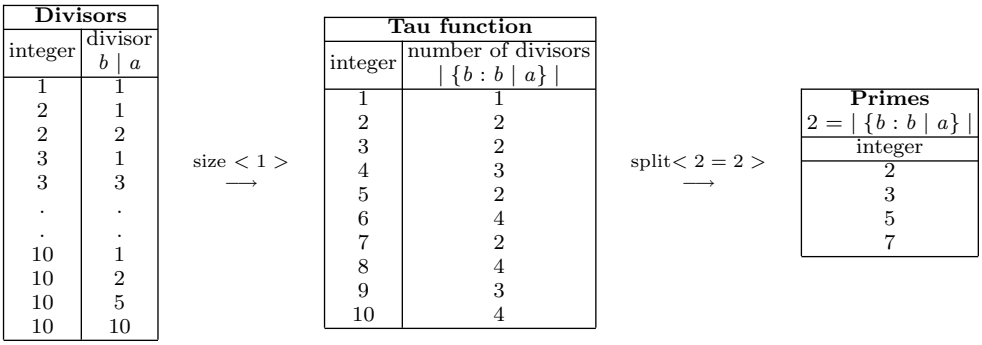


Fig. 3. Steps applied by HR to produce the concept of prime numbers.

which are used to solve the particular problem 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.
- The *size* rule: counts the number of tuples for each entry of a specified column of the data table.

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 that 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 [CBW00b] 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 ($b \mid a$ where b is a divisor of a), represented by a data table for a subset of integers (partially shown in Figure 3 for integers from 1 to 10). Then, HR would apply the size production rule with the parameterisation $< 1 >$. This means that

Prime numbers
$2 = \mid \{b : b \mid a\} \mid$
2
3
5
7

$\Rightarrow$

Non-square numbers
$\neg(exists\, b.(b \mid a \,\&\, b * b = a))$
2
3
5
6
7
8
10

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

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:

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

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

3. Automated theory formation for Event-B models with HR

In this section we describe how ATF, and in particular the HR system, can be used to discovery invariants within the context of Event-B. As mentioned in Section 2.2, HR requires domain information and an initial configuration as inputs to the theory formation process. The domain information consists of a list of concepts together with associated definitions and examples, while the configuration corresponds to the set of PRs that are to be enabled during theory formation. As shown in in Figure 5, concepts are extracted directly from an Event-B model, while examples are derived from simulation traces. Below we describe in detail these processes along with the selection of PRs. We use the refinement step presented in Section 2.1 as a running example.

3.1. Extracting concepts

As explained above (§2.2), there are two types of concepts in HR: those which are provided by a user at the beginning of a run – core concepts – and those developed by HR during the run – non-core concepts. We can further distinguish between core concepts which simply list the objects of interest (an extensional definition) and those which define features of the objects of interest. For instance, a data set about animals might contain an extensional definition of animal, such as *animal(dog), animal(dolphin), animal(turtle)*; and define features of the animals, such as their class, type of covering and number of legs: *class(dog,mammal), class(dolphin,mammal), class(turtle,reptile)*; *covering(dog,hair), covering(dolphin,none), covering(turtle,scales)*; and *legs(dog,4), legs(dolphin,0), legs(turtle,4)*. We categorise these respectively as concepts of type 1 and type 2, and non-core concepts as type 3.

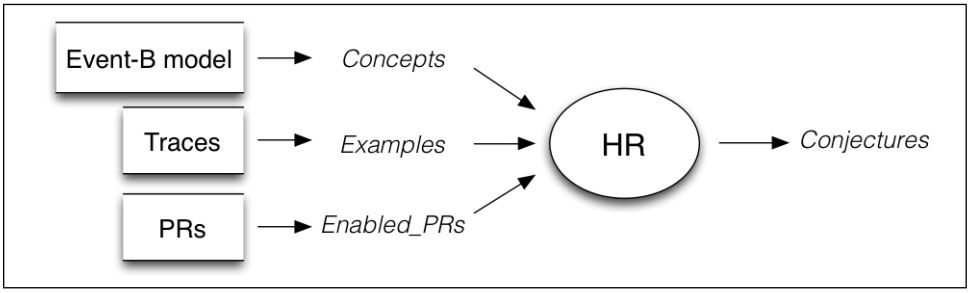


Fig. 5. Discovery of candidate invariants in Event-B models through HR.

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

Fig. 6. State in the abstract and concrete levels of the Mondex refinement step.

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

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

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

Within Event-B, the input concepts are defined by the state of the model which is represented via *carrier sets*, *constants* and *variables*. Carrier sets are considered to be T1 concepts because they represent the basic entities in an Event-B model, i.e. they are user defined types from which constants and variables can be formed. Variables are considered to be T2 concepts since they are defined in terms of the objects of interest, i.e. the carrier sets. Constants also represent T2 concepts unless they are members of a carrier set, in which case they are considered to be examples of the carrier set. More formally, the sets of T1 and T2 concepts are defined as follows:

$$conceptsT1(M) \triangleq carrierSets(M) \tag{2}$$

$$conceptsT2(M) \triangleq variables(M) \cup \{c \mid c \in constants(M) \wedge c \notin \bigcup_{s \in carrierSets(M)} s\} \tag{3}$$

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

To illustrate the process of concept extraction, consider again the refinement step from our running example (see Section 2.1). Figure 6 summarises the abstract and concrete state associated with this refinement step. The results of applying functions (2) and (3) to the abstract and concrete models are shown in Figures 7(a) and 7(b) respectively. Note that the constants associated with the concrete level, i.e. *IDLEF*, *EPR*, *etc*, are not identified as core concepts because they are members of the set *status*, as defined by the axiom:

$$partition(status, IDLEF, EPR, EPA, ABORTEPR, ABORTEPA, ENDF, IDLET, EPV, ABORTEPV, ENDT)$$

The next step involves translating the core concepts into HR definitions. An HR definition consists of a functor along with the type of its parameters. The required type information is given as invariants and axioms within a model. In the case of the Mondex example, the invariants shown in Figure 8 specify the types of the T2 concepts listed in Figure 7. From these invariants it is observed that the abstract variables *idleFP,*

$conceptsT1(abstractModel) = \{\text{purses}\}$
$conceptsT2(abstractModel) = \{\text{idleFP, eprP, epaP, endFP, idleTP, epvP, endTP, aborteprP, abortepaP, abortepvP}\}$

(a) Core concepts obtained from the abstract model.

$conceptsT1(concreteModel) = \{\text{status}\}$
$conceptsT2(concreteModel) = \{\text{statusF}\}$

(b) Core concepts obtained from the concrete model.

Fig. 7. Core concepts extracted from the Mondex refinement step.

Abstract invariants:

idleFP $\subseteq$ purses	epaP $\subseteq$ purses	abortepaP $\subseteq$ purses
idleTP $\subseteq$ purses	eprP $\subseteq$ purses	aborteprP $\subseteq$ purses
endFP $\subseteq$ purses	epvP $\subseteq$ purses	abortepvP $\subseteq$ purses
endTP $\subseteq$ purses		

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

Fig. 8. Type invariant for the Mondex core concepts.

eprP, etc. are subsets of *purses* while the concrete variable *statusF* is a function from *purses* to *status*. In terms of a HR definition, *idleFP* is represented as *idleFP(A)* where *A* denotes the type of *purses*. Similarly, *statusF* is represented as *statusF(A, B)* where *A* and *B* denote the types *purses* and *status* respectively. Figure 9 shows the HR definitions for all the core concepts that arise in the running example.

3.2. Generating examples of concepts

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

A trace represents a record of the behaviour of the system during the simulation, i.e. it contains the value of the domain data at each step. Drawing upon the Mondex refinement step, Figure 10 shows a fragment of a trace generated from a simulation run performed by ProB. Note that we have only shown the concepts that are needed for the generation of the gluing invariant (1). Note also that carrier set *purses* and variable *idleFP* denote abstract concepts, while the carrier set *status* and variable *statusF* denote concrete concepts. In order to be able to observe gluing invariants we must be able to link the abstract and concrete states. To achieve this, *state* is added as a core concept to the domain of an Event-B model.

In terms of generating examples, where a carrier set is finite all members of the set are included. For

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

Fig. 9. Definitions of the Mondex core concepts.

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

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

Fig. 10. An example animation trace.

instance, ProB automatically selects the constants *IDLEF, EPR, EPA,* etc. as the examples of *status*. In the case of carrier sets such as *purses*, where there are no axioms defining membership, ProB dynamically generates a list of arbitrary members using the name of the carrier set as a prefix. To illustrate, for *purses* ProB generates the list *purses1, purses2, purses3, purses4* and *purses5*.

Each step within the simulation trace represents the execution of an event. While the event itself is not part of the trace, its effect on the value of the variables is recorded. Note that the symbol '–' is used to denote an undefined variable. Simulations can be generated randomly or through the use of test case generators. The quality of the invariants depends on the quality of the simulations, so the use of test case generators is highly recommended whenever possible. For the Rodin toolset, no test case generator is available. The simulations used in our experiments are randomly generated as the development of a test case generator is outside of the scope of our work.

3.3. Constructing data tables

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

$$\begin{aligned} DataTable(x) \triangleq\ & \text{if } isa\_conceptT1(x) \text{ then } \{e \mid e \in x\} \\ & \text{else } \{< s, e > \mid s \in state \ \wedge \ e \in trace(s, x)\} \end{aligned} \tag{4}$$

where $isa\_conceptT1(x)$ is true iff x denotes a T1 concept, and $trace(s, x)$ computes the set of examples e for concept x in state s. Note that in the case of a T1 concept, the associated data table simply enumerates the elements of the concept. For example, applying (4) to the T1 concepts given in Figure 10 yields:

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

The corresponding data tables are given in Figure 11. Note that a data table for *state* is always required, as it is required in order to link abstract and concrete perspectives. In the case of a T2 concept, the associated data tables corresponds to a set of tuples that associate examples of the concept with the states in which they appear within the trace. To illustrate, the application of (4) to the T2 concepts given in Figure 10

purses
purses1
purses2
purses3
purses4
purses5

status
IDLEF
EPR
EPA
ABORTEPR
ABORTEPA
ENDF
IDLET
EPV
ABORTEPV
ENDT

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

Fig. 11. Example data tables for T1 concepts.

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

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

Fig. 12. Example data tables for T2 concepts.

yields:

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

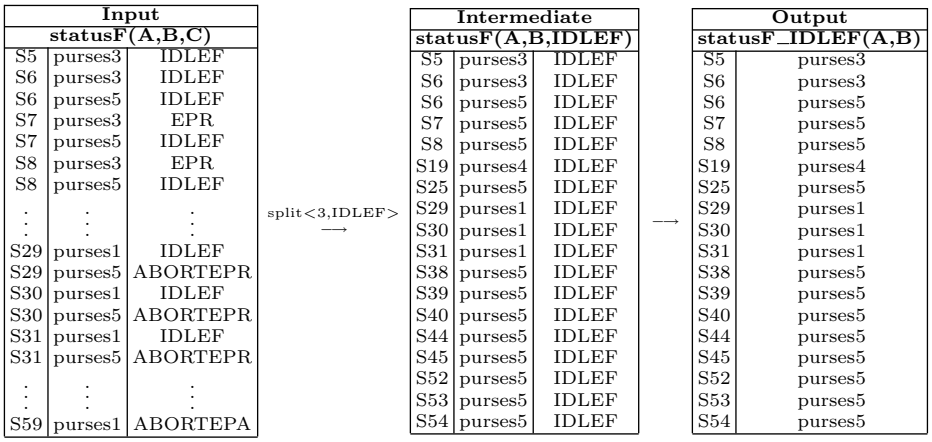
The corresponding data tables are shown in Figure 12.

3.4. Selecting PRs and running HR

Once provided with data tables, HR applies all possible combinations of concepts and PRs in order to generate new concepts and form conjectures. HR currently contains a set of 22 PRs. In our work we have

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

Table 1. Compatibility between production rules and formula operators.



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.

Fig. 13. Concept of purses whose status is *IDLEF*.

focused on 7, which are relevant to the Event-B formalism. The remaining PRs are mainly domain specific. For instance, HR contains PRs specialised to generate integer sequences and to handle operations over floating point numbers. We disable these special purpose PRs from the invariant discovery process. Table 1 shows the correspondence between the 7 selected PRs and the Event-B operators. However, as will be explained in Section 7, we believe that there is scope for new PRs that address aspects of the formalism not currently covered.

In order to illustrate HR's theory formation mechanism, consider again the data tables given in Figures 11 and 12. Starting with these tables, and after 433 steps of a 1000 step theory formation run, HR forms the following conjecture:

$$\forall A, B.\big(state(A) \wedge purses(B) \wedge idleFP(A,B) \Leftrightarrow status(IDLEF) \wedge statusF(A,B,IDLEF)\big) \tag{5}$$

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

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

which is gluing invariant (1). The discovery of this conjecture involves observing that column 2 of data table **idleFP(A,B)** is identical to the rows of data table **statusF(A,B,IDLEF)**. The *split* PR plays a central role in this discovery as illustrated in Figure 13. Within the theory formation process, when the *split* PR is applied with the parameters <3, IDLEF>, then a specialisation of concept *statusF* is formed, i.e. a data table is formed by extracting the tuples whose third column matches the parameter *IDLEF*. Note that since the third column is the same for all tuples, this column is removed from the output concept.

Immediately after the generation of new concepts, HR looks for relationships with other existing concepts. As shown in Figure 14, HR discovers that the new concept, i.e. *status_IDLEF*, has the same list of examples as concept *idleFP*, giving rise to conjecture (5).

3.5. Challenges

In general, using HR for the discovery of invariants of Event-B models presented us with three main challenges:

1. As highlighted in our running example, HR produces a large number of conjectures – in our experiments the set of generated conjectures was in the range of 3000 to 12000 conjectures per run – from which

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

$\Leftrightarrow$

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

Fig. 14. Formed equivalence conjecture.

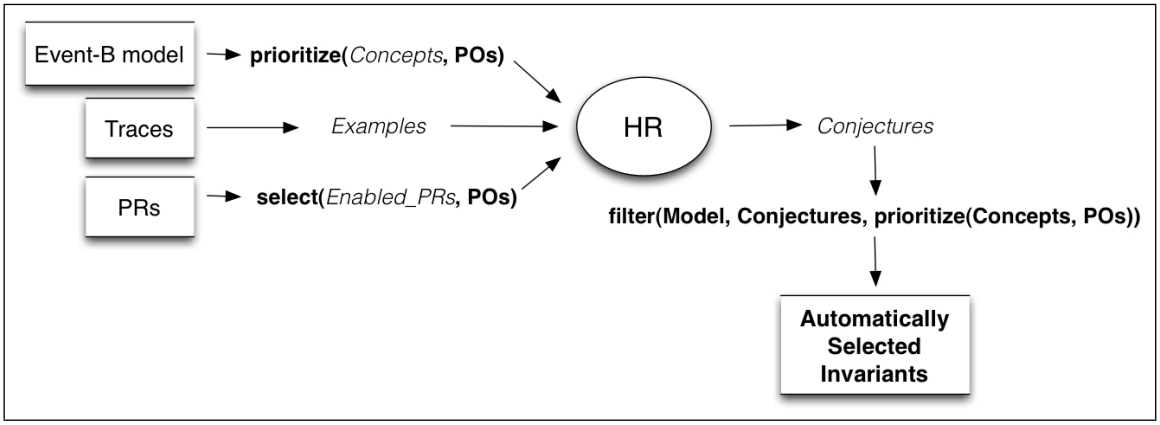
only a very small set represent interesting invariants of the system. Our main challenge was to find a way of automatically selecting the conjectures that are interesting for the domain among the conjectures obtained from HR; in this way, only a handful of candidate invariants is presented to the user instead of a large set of uninteresting conjectures.

2. The HR theory formation mechanism consists of an iterative application of production rules over all concepts in the theory. In order for HR to perform an exhaustive search, all possible combinations of production rules and concepts must be carried out. However, there is not a fixed number of theory formation steps set up for this process, since this varies depending on the domain, i.e. some domains need more theory formation steps than others. This represented a challenge for the use of HR in the discovery of invariants since it was possible that an invariant had not been formed only because not enough steps had been applied, and performing an exhaustive search would give rise to an unmanageable number of conjectures.
3. Some production rules are more effective in certain domains than others. Selecting the appropriate production rules results in the construction of a more interesting theory. For instance, if we are looking at a refinement step in an Event-B model that introduces a partition of sets we expect the new invariants to define properties over the new sets; therefore, production rules such as the *arithmetic PR* will not be of much interest in the development of the theory associated to the refinement step. Automatically selecting appropriate production rules requires knowledge about the domain; therefore, a technique was needed in order to perform this selection.

In addressing these challenges, we developed a set of heuristics which automatically constrain both the configuration of HR and the selection of conjectures that arise from a theory formation run. In the following section we describe these heuristics and their application in detail.

4. Heuristic Approach

As illustrated above, the use of HR in discovering Event-B invariants can involve significant user interaction. In particular, the user must supply domain information about their models, as well as the set of the PRs that are to be enabled during theory formation. For example, recall in Section 3.3 where invariant (1) was discovered by an application of the split PR to the concept *statusF* using the parameter <3, IDLEF>. Without guidance, HR's search for such parameter settings makes invariant discovery infeasible. To achieve such guidance, we have developed a set of heuristics. Figure 15 provides a high-level picture of how these heuristics relate to the basic HR data flow diagram given in Figure 5 for invariant discovery. Central to the design of our heuristics is the observation that proof-failure analysis provides insights into the structure of missing invariants. This is reflected in the fact that each heuristic is parametrised by the POs associated with a failed refinement step. Note that the heuristics are used both in configuring HR, i.e. *prioritise* and *select*, as well as filtering the conjectures that are generated by HR, i.e. *filter*. Below we provide definitions



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

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

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

Fig. 15. Approach for the automatic discovery of invariants.

and explanations for each heuristic, focusing first on the *Configuration Heuristics* (CH) in Section 4.1 and then the *Filtering Heuristics* (FH) in Section 4.2.

4.1. HR Configuration Heuristics

Each new concept formed by HR can be combined with any existing concept; furthermore, PRs can be applied with different parameterisations. This means that the theory formation process can lead to a combinatorial explosion [CBW00a]. HR uses an *agenda* mechanism to organise how concepts are explored during the theory formation process. By prioritising concepts, we use the CH heuristics to influence this agenda mechanism and constrain the applicable PRs during theory formation. We use two overall heuristics, i.e. CH1 and CH2, when configuring HR for a given Event-B refinement step:

CH1. *Prioritises core and anticipated non-core concepts that occur within the failed POs*
Prioritise concepts that occur within the failed POs: Highest priority is given to core concepts that occur within goals of the failed POs, followed by core concepts that occur within the hypotheses. Other core concepts are dealt with next. Finally, compound expressions that occur within the goal (and hypotheses) which represent potential non-core concepts are identified, by finding expressions within the POs which can be replicated via the application of PRs. For instance, if a PO contains the expression $a \vee b$, where a and b are core concepts, the *disjunct* PR can be applied, with the two core concepts as inputs, to replicate the expression within HR. Therefore, $a \vee b$ represents a non-core concept. We define these as "anticipated" non-core concepts. Note that we assume that the input POs are well defined, since they are generated from the Rodin toolset; therefore, no invalid expressions would arise when identifying non-core concepts from the predicates of a PO. The implementation details of how these concepts are extracted from the POs is provided in Section 6.

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

where:

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

CH2. *Select the subset of the enabled PRs that are most relevant to the given failed POs, i.e.*

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

where:

operators(POs): denotes the set of operators that occur within the predicates associated with POs.
related_ops(PR): denotes the set of operators that are associated with PR (see Table 1).

Note that 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.

As will be shown in Section 7, the empirical evidence we have gathered so far supports our rationale for both the configuration heuristics.

4.2. Conjecture Filtering Heuristics

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

$$
\begin{aligned}
&\mathit{filter}(\mathit{Model}, \mathit{Conjectures}, \mathit{POs}, \mathit{PriConcepts}) \ \triangleq \\
&\quad \{c \mid c \in \mathit{filter}_5(\mathit{Model}, \mathit{POs}, \mathit{filter}_4(\mathit{POs}, \mathit{filter}_3(\mathit{filter}_2(\mathit{filter}_1(\mathit{Conjectures}, \mathit{PriConcepts})))))\}
\end{aligned}
$$

Each of the 5 filters are defined below:

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

$$
\begin{aligned}
&\mathit{filter}_1(\mathit{Conjectures}, \mathit{PriConcepts}) \triangleq \\
&\quad \{c \mid c \in \mathit{Conjectures} \wedge \\
&\qquad (\exists \ \mathit{pc} \in \mathit{PriConcepts} \ .(c = (\_ \Rightarrow \_) \ \vee \ c = (\_ \Leftrightarrow \_) \vee c = \neg\, \exists(\_)) \ \wedge \ \mathit{occurs\_in}(\mathit{pc}, c))\}
\end{aligned}
$$

where the predicate *occurs_in(pc, c)* holds if and only if *ps* occurs in *c*. Note that only implication, equivalence and non-existence conjectures which relate to prioritised concepts are selected.

FH2. *Select conjectures where the sets of variables occurring on the left- and right-hand sides are disjoint,*

i.e.

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

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

FH3. *Select only the most general conjectures, i.e.*

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

FH4. *Select conjectures that discharge the failed POs, i.e.*

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

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

FH5. *Select conjectures that minimise the number of additional proof-failures that are introduced.* Overcoming a proof-failure potentially leads to new proof-failures. That is because in order to ensure that the system is consistent with a new invariant, new POs are generated. This may give rise to proof failures when additional properties are required in order to prove the model does not violate the new invariant. Here we select candidate invariants that minimise the number of new failures, i.e.

$$\begin{array}{l} \mathit{filter}_5(\mathit{Model}, \mathit{POs}, \mathit{Conjectures}) \triangleq \\ \quad \{c \mid c \in \bigcup_{\mathit{po} \in \mathit{POs}} \mathit{minFailedPOs}(\mathit{Model}, \mathit{po}, \mathit{Conjectures})\} \end{array}$$

where

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

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

The application of filtering heuristics described above may involve iteration. That is, if focusing on concepts within the goal of a failed PO does not generate suitable invariants, then the filtering heuristics are reapplied to the concepts appearing with the hypotheses. In other words, in the first iteration, the parameters of the filtering heuristics are the prioritised core and non-core concepts that appear within the goal of the failed POs and the conjectures associated to them. In the second iteration, the heuristics are parameterised with the core and non-core concepts that appear within the hypotheses of the failed POs and the conjectures associated to them.

5. Mondex invariant discovery revisited

To illustrate the application of our heuristics, we return to the running example, and the discovery of the missing gluing invariant:

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

Fig. 16. Failure resulting from a missing gluing invariant.

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

Without this invariant, an unprovable guard strengthening (*GRD*) PO[3] is generated for the event *StartFrom*. The unprovable PO together with the abstract and concrete versions of the event *StartFrom* are shown in Figure 16. Inspection of the PO shows that the abstract guard *p1* ∈ *idleFP* (goal) is not implied by the concrete guards (hypotheses).

The application of the configuration heuristics starts with CH1, i.e. the prioritisation of core and non-core concepts within the failed PO. The following core concepts are identified within the goal and hypotheses of the failed PO:

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

Note that the identifiers *p1* and *t*, which appear within the failed PO, are not considered as core concepts as they are parameters, i.e. not variables, constants or sets. Furthermore, we identify the non-core concept:

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

This non-core concept is identified because of the occurrences of constant *IDLEF*, which is a split value, and *statusF*, which is a core concept in hypothesis:

$$\text{p1} \mapsto \text{IDLEF} \in \text{statusF}$$

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

Turning next CH2, the following PRs are selected for the invariant discovery process:

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

As mentioned previously, the compose, disjunct and negate PRs are always selected by CH2. The split PR is selected when any reference to a member of a finite set occurs. In the context of the example, the constant IDLEF, which is a member of the finite set *status*, triggers the selection of the split PR.

Thus, the split PR is applied over the finite set *status* and the values to split are all the members of the set, i.e.: *IDLEF, EPR, EPA, ABORTEPR, ABORTEPA, ENDF, IDLET, EPV, ABORTEPV* and *ENDT*.

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

Next we follow with the application of the filtering heuristics focusing on the conjectures that relate to the prioritised core and non-core concepts – FH1 prunes the candidate invariants to give:

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

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

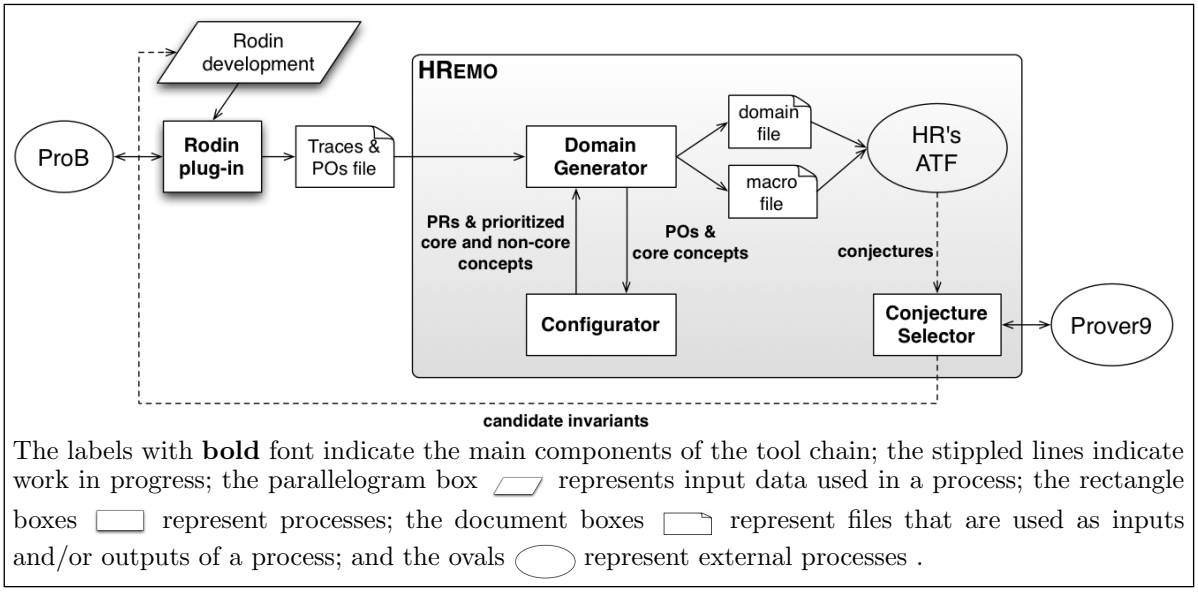


Fig. 17. Tool-chain architecture

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

1 equivalence, 2 implications and 79 non-exists conjectures

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

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

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

1 equivalence, 0 implications and 0 non-exists conjectures

Note that in this example a single conjecture remains. Furthermore, this conjecture represents the required invariant, i.e. (5), and does not introduce any additional proof-failures.

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

6. Invariant Discovery Implementation

In this section we describe a prototype implementation of our heuristic approach to invariant discovery. The tool architecture is shown in Figure 17. 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 the HR domain description. Our heuristics were mechanised via an extension to HR; i.e. HRemo. Both the HR tool and the Rodin toolset are implemented in Java; therefore, the implementation effort was also carried out in Java. Below we describe each of the major components of the tool.

6.1. 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 extracting domain information from an Event-B model, i.e. the animation traces and failed POs, and sending it to HRemo.

As shown in Figure 17, the plug-in receives as input a Rodin development. The user selects an Event-B machine in the associated refinement chain for which the invariant discovery analysis should be performed – note that only one step of the refinement is handled at a time. ProB is then invoked in order to run a random simulation of the selected refinement step. Currently, the plug-in sets up ProB to generate 100 random steps – selecting fewer steps may not provide a representative sample of the behaviour of the system and selecting a larger number may affect the performance of HR; for instance, a PR may not be applicable when the number of tuples in the output data tables exceeds the maximum allowed by the PR.

When the simulation finishes, the plug-in extracts the structure of the model, i.e. variables, constants and sets, the examples from the animation traces and the failed POs. This information is then output into an xml file: the *traces and POs file*. 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. The xml file produced by the plug-in contains:

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

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

6.2. Domain Generator

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

The traces contained in the file are transformed into background concepts in HR; i.e. the definition and data tables of each variable, constant and set are built as explained in Section 3. Parsing the POs consists in translating the tree-like representation explained in the previous section into a similar tree-like representation within HRemo; i.e. classifying each predicate in the PO as either a binary, unary or literal formula. After the POs are parsed, the domain generator uses the *configurator* to analyse the failed POs – this is where the configuration heuristics are applied. The configurator determines the order of the concepts in the domain file and the non-core concepts that are to be forced into the domain, as well as which production rules are to be specified in the macro file. Details of this analysis are given in the next section.

6.3. Configurator

The *configurator* receives the parsed POs and the core background concepts from the domain generator. Its role is to apply the configuration heuristics in order to make the selection of the prioritised core and non-core concepts as well as the production rules from the failed POs. The pseudo-code for the identification of core and non-core concepts is given in Algorithm 1. The analysis of this process is as follows:

1. The configurator analyses the goals and then the hypotheses within the failed POs. This is because 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. Through the recursive procedure described in Algorithm 1, the configurator examines each formula to decide whether or not there exists a core or a non-core concept. This decision is made based on the following parameters:
   (a) A core concept is identified if a literal formula has been found (Line 2), and the identifier associated to the formula represents a variable or constant from the domain background concepts (Line 4).

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

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

(b) A non-core concept is a combination of variables and constants that can be replicated in HR through the use of the PRs. In order to identify non-core concepts from a formula we analyse the compatibilities between formula operators and PRs. These compatibilities were introduced in Table 1. A binary or unary formula is a valid non-core concept if its operator is compatible with a PR and its sub-formulas are themselves valid formulas. Therefore, for unary formulas (Line 7), the sub-formula is recursively analysed (Line 10) and if the operator is compatible with a PR and the sub-formula is valid (Line 11) the formula is added as a non-core concept. Likewise, for binary formulas (Line 14), the left and right sub-formulas are recursively analysed through the function (Lines 18 and 19) and if the operator is compatible with a PR and the left and right sub-formulas are valid (Line 20) the formula is added as a non-core concept.

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

i 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, i.e. condition in Line 20 does not hold;

ii the formula contains valid split values (Line 26). Algorithm 2 shows the pseudocode of the procedure that identifies split values from a formula. A valid split value is an identifier from a literal formula (Line 4) that does not represent a background concept (Line 5) but that is a member of

a domain set (Line 7); e.g. the value *green* is a member of the domain set *Color*; therefore, *green* is a valid split value; and

iii the formula contains a core concept that is compatible with the split value. A core concept is compatible with a split value if one of its parameters could be assigned the split value (Algorithm 1, Line 30); e.g. the core concept *ml_tl*, which denotes a traffic light, could be assigned the value *green*.

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

```
1  function getSplitValues(formula)
2      splitValues ← split values found in formula
3      if formula is a literal formula then
4          literal ← get identifier from formula
5          if literal not a domain variable AND literal not a domain constant then
6              for all set in domain sets do
7                  if literal is an element of set then
8                      add literal to splitValues
9      else if formula is a unary formula then
10         uniFormula ← get unary sub-formula from Formula
11         splitValues ← splitValues + getSplitValues(uniFormula)
12     else if formula is a binary formula then
13         leftFormula ← get left sub-formula from Formula
14         rightFormula ← get right sub-formula from Formula
15         splitValues ← splitValues + getSplitValues(leftFormula)
16         splitValues ← splitValues + getSplitValues(rightFormula)
17     return splitValues
```

The same methodology is followed in order identify the PRs that are used during the search for conjectures within the theory formation process. That is, when an operator is found that is compatible with a PR, then the PR is enabled for the search.

6.4. Conjecture selector

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

1. The identification of the *predecessors* of a concept. The predecessors are the core concepts involved in the construction of a concept generated through the PRs. Algorithm 3 shows the pseudocode for the identification of the predecessors of a concept. If the concept represents a core concept (Line 2) its id is returned; otherwise, the predecessors of the concept are the predecessors of its parent concepts; i.e. the concepts that were immediately used together with a PR in order to generate the concept within the theory (Lines 6-8).
2. The identification of *alternative conjectures*. That is, when HR finds two equivalent concepts only one of them is kept in the theory since both concepts will derive the same conjectures. This resulted in some interesting implication conjectures not being identified because they were "hidden" to us by the equivalences previously formed by HR[4]. In particular, interesting implications were not identified when a

[4] This problem only arises with implication conjectures. This is because as prioritised concepts are always kept in the theory – since they are generated before the theory formation process starts, i.e. before HR is run – all concepts equivalent to them are identified. Furthermore, HR does not form equivalences with concepts that have empty data tables; therefore, each concept with an empty data table forms a non-existence conjecture.

Algorithm 3 Identification of the predecessors of a concept.

```
1  function PREDECESSORS(concept)
2      if concept is a core concept then
3          return id of concept
4      else
5          preds ← stores the predecessors of concept
6          parentConcepts ← get parent concepts of concept
7          for all c in parentConcepts do
8              preds ← preds + predecessors(c)
9          return preds
```

prioritised concept appeared in both sides of the conjecture. For instance, in level four of the refinement chain of the Mondex example, the following implication conjecture is generated by HR:

$$\forall\, a, b \cdot pending(a, b) \Rightarrow (idle(a, b) \vee pending(a, b) \vee epa(a, b))$$

where *pending, idle* and *epa* represent core concepts. As can be observed, the concept *pending* occurs on both sides of the implication; therefore, the conjecture would be discarded by our heuristics. However, by exploring the equivalence conjectures, we found that the concept in the right hand side of the implication is equivalent to the concept:

$$idle(a, b) \vee epv(a, b) \vee epa(a, b).$$

Based on this equivalence, we can replace the right hand side of the original implication conjecture, with the equivalent concept. This produces a new implication conjecture, i.e.:

$$\forall\, a, b \cdot pending(a, b) \Rightarrow (idle(a, b) \vee epv(a, b) \vee epa(a, b))$$

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

Algorithm 4 Identification of alternative conjectures of a prioritised concept.

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

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

1. Algorithm 5 shows the pseudocode for the selection of the conjectures associated with prioritised core

Algorithm 5 Selection of conjectures associated to prioritised core and non-core concepts.

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

10 function getNonExistenceConjectures(prioritisedConcept)
11     conjectures ← stores the non-existence conjectures of prioritisedConcept
12     nonExists ← get non-existence conjectures from theory
13     for all nEx in nonExists do
14         nExConcept ← get concept from nEx
15         if prioritisedConcept is in predecessors(nExConcept) then
16             add nEx to conjectures
17     return conjectures

18 function getImplications(prioritisedConcept)
19     conjectures ← stores the generalisations of prioritisedConcept
20     concepts ← get all concepts from theory
21                                                  ▷ Get the generalisations.
22     for all c in concepts do
23         generalisations ← get generalisations from c
24         if prioritisedConcept in generalisations then
25             if prioritisedConcept is in predecessors(c) then
26                 altConjs ← alternativeConjectures(prioritisedConcept, c)
27                 conjectures ← conjectures + altConjs
28             else
29                 imp ← c ⇒ prioritisedConcept
30                 add imp to conjectures
31                                                  ▷ Get the implications
32     implications ← get implication conjectures from theory
33     for all imp in implications do
34         left ← get left concept from imp
35         right ← get right concept from imp
36         if prioritisedConcept = left then
37             if prioritisedConcept is in predecessors(right) then
38                 altConjs ← alternativeConjectures(prioritisedConcept, right)
39                 conjectures ← conjectures + altConjs
40             else
41                 add imp to conjectures
42         else if prioritisedConcept = right then
43             ...                                  ▷ similar analysis than previous case.
44     return conjectures
```

and non-core concepts. Finding the equivalence conjectures is a simple process. For each equivalence in the theory (Line 4), if the left- or the right-hand side concept of the conjecture is equal to the prioritised concept (Line 7) the equivalence is added to the set of interesting conjectures (Line 8). Regarding non-existence conjectures (Line 10), if a prioritised concept occurs as a predecessor of an non-existence conjecture (Line 15), then the non-existence conjecture is added to the set of interesting conjectures (Line 16). Identifying the implication conjectures requires:

(a) Finding the concepts for which the prioritised concepts are *generalisations*. HR defines a concept *c1* as a generalisation of a concept *c2* if the definition of *c1* is contained within the definition of *c2*. In other words, generalisations are identified by comparing the definitions of the concepts, while the implication conjectures are identified by comparing their data tables. Thus, if a prioritised concept is a generalisation of a concept c (Line 24) and the prioritised concept is a predecessor of c (Line 25), then alternative conjectures between the prioritised concept and c are found (Line 26). However, if the prioritised concept is not a predecessor of c, i.e. the condition in Line 25 fails, then an implication conjecture of the form $c \Rightarrow$ *prioritisedConcept* is created (Line 29) and added to the collection of interesting conjectures (Line 30).

(b) Finding the implication conjectures associated to the prioritised concepts. If the left-hand side concept of an implication is equal to the prioritised concept (Line 36) and if the prioritised concept is a predecessor of the right-hand side concept (Line 37), then alternative conjectures of the prioritised concept and the right-hand side concept are found (Line 38). If both sides of the implication are disjoint, i.e. condition in Line 37 fails, then the implication is added to the set of interesting conjectures (Line 41). A similar analysis is carried out if the prioritised concept is equal to the right-hand side of the implication (Line 42).

2. Next, the disjoint conjectures are selected. Algorithm 6 shows the pseudocode to identify if a conjecture is disjoint. If the conjecture is an implication or an equivalence (Line 2), and if left- and right-hand sides have a core concept in common which represents a variable of the domain (Line 8) then the conjecture is not disjoint (Line 9). On the other hand, a non-existence conjecture (Line 10) is not disjoint if there exists a core concept that is a variable which occurs more than once in its predecessors (Lines 15 and 16). The given conjecture is disjoint otherwise (Line 18).

Algorithm 6 Selection of disjoint conjectures.

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

3. Prover9 [McC10] is an automated theorem prover for first-order and equational logic, and is the successor of Otter [McC03]. Prover9 has the usual limitations of all theorem provers for first order theories; for instance, incompleteness. Also, it lacks support for interactive proofs. However, Prover9 counts with

several tools that support it, such as counterexamples checkers, tools that produce more detailed proofs or that export proofs to other languages, etc.
While HR uses Prover9 to prove/disprove conjectures, we use it to choose the most general conjectures. Given two conjectures *conj1* and *conj2*, if *conj2* is logically implied by *conj1*, *conj2* can be removed from the set of candidate invariants.

4. Selecting the conjectures that discharge the failed POs (FH4) and produce fewer extra failed POs (FH5) are the final two steps of the the filtering process. We have implemented heuristics FH1, FH2 and FH3; however, although we believe the approach can be applied to different formalisms, the implementation of heuristics FH4 and FH5 are language- and tool-dependent; this is due to the fact that the generation of POs in a formal model requires specialised mechanisms that are particular to each formalism. Therefore, heuristics FH4 and FH5 are currently performed manually. This involves manually translating each conjecture generated up to heuristic FH3 into Event-B and introducing one by one to the Event-B model in the Rodin toolset. Then, a record is kept of the failures addressed by each conjecture and when all the conjectures have been analysed, the recorded information is compared to select the final set of invariants. As can be inferred from this, the implementation of heuristics FH4 and FH5 for the context of Event-B would involve automatically translating the conjectures from the format given by HR into the Event-B formalism. This translation will be pursued in future work.

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

7. Experimental results

The experiments[5] 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. As mentioned in Section 2.1, this development is composed of one abstract model and nine refinement steps. We applied our invariant discovery technique over each of the these refinement steps.
3. *Location access controller system [Abr10]:* a model of a system that controls access to the rooms within a building.

[5] The files and Rodin developments related to the experiments can be found in http://www.macs.hw.ac.uk/remo/ under "HREMO Experimental Results". In order to reproduce the experiments, please contact the authors to get a copy of HREMO and the traces generator plug-in for the Rodin toolset.

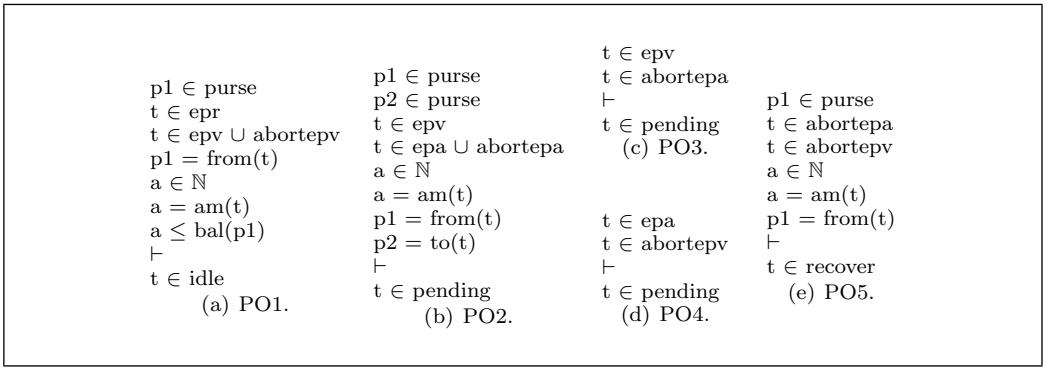


Fig. 18. First set of failed POs.

4. *Flash-based file system [Dam10]:* a model of a flash-based file system that allows a user to read, write and erase information from a flash disk. This development consists of two models; one that handles the logical operations (software) and the other that handles the physical operations (hardware). We targeted the latter, which involves 5 refinement steps, and models the reading, writing and erasing of the physical pages within a flash disk.

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 re-constructed the Mondex system in Event-B based upon the development presented in [BY08]. In the following section we present the results obtained by applying our approach to the refinement of level three of the Mondex system. Furthermore, we show that these results are similar to the ones obtained through the interactive development [BY08]. Note that in Sections 7.1.1 and 7.2 we summarise the results we achieved across the whole refinement chain of our re-construction of the Mondex system model and the flash file system case study.

7.1. 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. That is: states *idleF, epr, epa, abortepr, abortepa* and *endF* for the source side of a transaction, and states *idleT, epv, abortepv* and *endT* for the target side. These states model the communication between the source and the target sides during a transaction; e.g. expecting request (epr), expecting value (epv), etc[6]. In order to evaluate our approach, we introduced the model in level 4 with only basic typing invariants. The absence of the rest of the invariants produces the failed POs shown in Figure 18. In order to discharged these failed POs, the gluing invariants that relate the abstract states with the states introduced in the refinement are needed.

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

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

Note that *t*, *a*, *p1* and *p2* do not represent core concepts since they are parameters of events in the model. Moreover, from the analysis of the predicates in the failed POs, 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. Figure 19 illustrates how $epv \cup abortepv$ is identified as a non-core concept from the

[6] Further details about the model of the Mondex system can be found in [BY08].

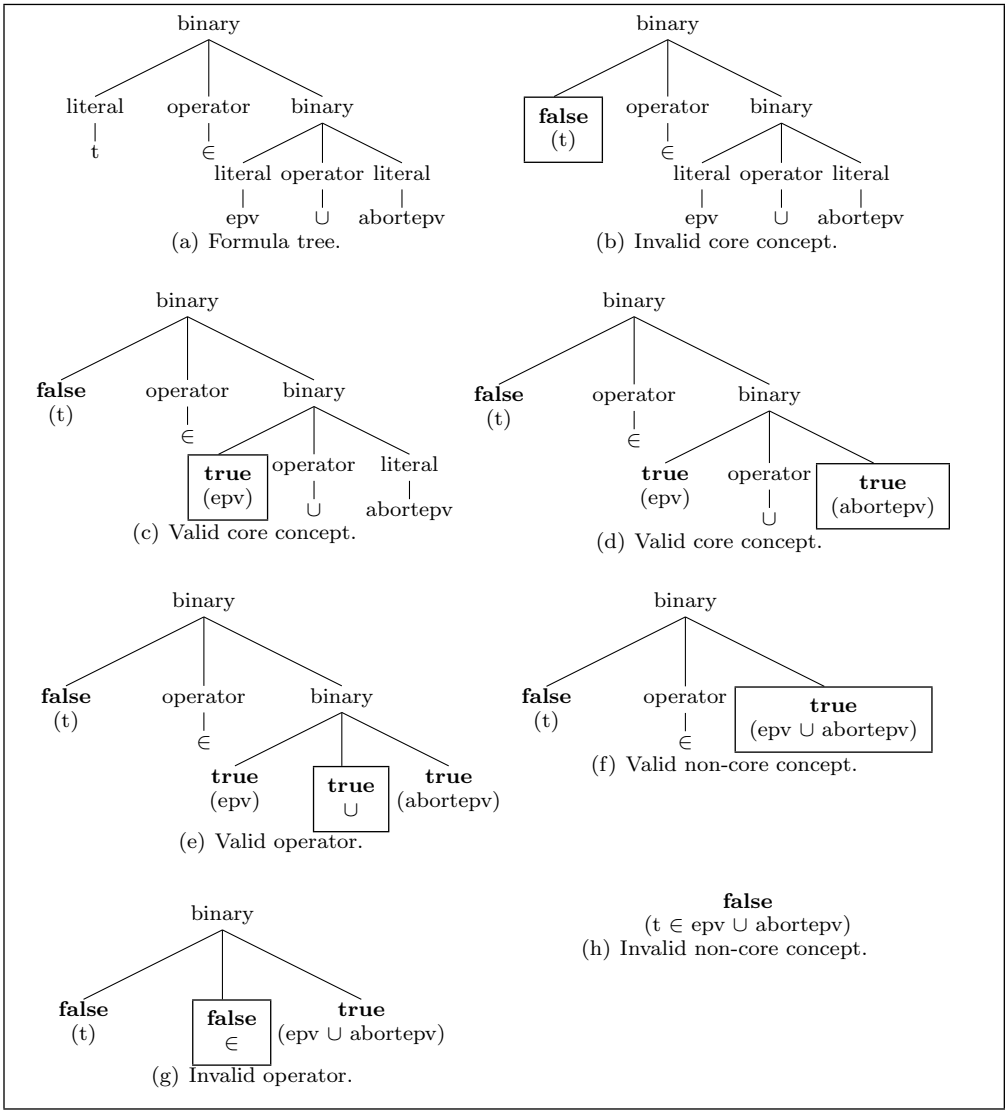


Fig. 19. Identifying core and non-core concepts from formula $t \in epv \cup abortepv$.

formula $t \in epv \cup abortepv$ (based on Algorithm 1 described previously in Section 6.3). The formula tree is transversed until finding the leaves (Figure 19(a)), then each node is analysed returning true if the node is a valid formula, i.e. a core or a non-core concept, or a compatible operator. It returns false otherwise. Note that the left sub-formula is not a valid core or non-core concept (Figure 19(b)). This is because t does not represent a concept in the domain, i.e. it represents a parameter of the event associated with the failed PO. On the contrary, *epv* and *abortepv* are valid core concepts (Figure 19(c) and Figure 19(d)) as they are variables in the domain. Furthermore, $\cup$ is a valid operator (Figure 19(e)) since it is compatible with the *disjunct* PR as stated in Table 1. This results in the right sub-formula being identified as a valid non-core concept (Figure 19(f)). Finally, the operator $\in$ is not valid (Figure 19(g)) as it is not compatible with any PR. For this reason, and because the left sub-formula is also invalid, the formula as a whole is not considered as a valid non-core concept (Figure 19(h)). A similar analysis is carried out over formula $t \in epa \cup abortepa$.

The process continues with the selection of the PRs. Based on the failed POs shown in Figure 18, the following PRs are selected for the search:

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

Table 2. Results of the application of filtering heuristics FH1, FH2, FH3 and FH4.

Concept	Heuristic	Equivalences	Implications	Non-exists
idle	FH1	7	27	24
	FH2	0	27	24
	FH3	0	16	17
	FH4	0	3	0
pending	FH1	6	27	35
	FH2	0	27	35
	FH3	0	8	26
	FH4	0	2	0
recover	FH1	9	51	41
	FH2	2	51	41
	FH3	2	3	30
	FH4	1	0	0

Table 3. Conjectures obtained after applying filtering heuristic FH4.

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

The compose, disjunct and negate PRs are always used in the search as stated in heuristic CH2. The numrelation production rule is selected because of the occurrence of operator $\leq$ in hypothesis $a \leq bal(p1)$ within PO1. After the configuration heuristics have been applied HR is run for 1000 steps, giving rise to 7296 conjectures.

The filtering heuristics are now applied to this set of conjectures. Heuristic SH1 suggests looking at the prioritised concepts associated with the goals of the failed POs. From the goals of the POs shown in Figure 18, we identified the concepts *idle, pending* and *recover*. Thus, we look for the conjectures associated to each of these concepts. The results of applying FH1 are shown in Table 2, along with the results obtained for heuristics FH2, FH3 and FH4 for each of the selected concepts.

As can be observed, after applying the four initial filtering heuristics we have narrowed the set of selected conjectures to six: three implications involving the concept *idle*, two implications for concept *pending* and one equivalence about the concept *recover*. These conjectures are presented in Table 3[7].

Heuristic FH5 is the final step in the discovery process, and selects the conjectures that produces the smallest number of new PO failures. The correspondence between the conjectures and the failures they help overcome is presented in Table 3 as well as the number of extra failed POs generated when they are introduced into the model.

As can be observed, the three conjectures associated with concept *idle*, i.e. conjectures 1, 2 and 3, discharge PO1. However, two of them each generate two new failed POs, while the other conjecture only generates one extra failure. Thus, conjecture 2, which produces the least number of failures, is presented as a candidate invariant. Regarding the two conjectures associated with concept *pending*, one of them discharges PO2 and PO3 and produces two new failed POs, while the other one discharges PO4 but produces three new failed POs. As there are no other conjectures that help overcome the failures produced by PO2, PO3 and PO4, both conjectures are suggested as candidate invariants. Finally, the equivalence conjecture associated with concept *recover* discharges PO5 and it does not produce any extra failures, so this conjecture is also suggested as a candidate invariant. Thus, the candidate invariants obtained from the first iteration of HREMO over the third refinement of the Mondex system are shown in Figure 20.

After the new set of invariants is introduced into the model, six new PO failures are generated. The new set of failed POs are shown in Figure 21. These POs raise failures related to the preservation of the invariants found in the first iteration. A second iteration of the invariant discovery process is then performed based on

[7] Note that we have given the equivalent set theoretic representation of the conjectures instead of using the universally quantified format provided by HR. This is because some experiments, for instance the development of the Mondex system carried out in [BY08], have shown that the automatic provers do better with quantifier-free predicates.

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

Fig. 20. Candidate invariants obtained from first iteration.

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

(a) PO6.

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

(b) PO7.

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

(c) PO8.

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

(d) PO9.

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

(e) PO10.

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

(f) PO11.

Fig. 21. Second set of failed POs.

the new set of failed POs. The application of the configuration heuristics results in the following prioritised core and non-core concepts:

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

and in the selection of the following PRs:

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

As with the first iteration, we apply the filtering heuristics first to the core concepts identified in the goals of the failed POs. Therefore, the search for invariants is focused in concepts *epv, epa, abortepa, pending, abortepv, idleF, epr* and *idle*. A summary of the result of the application of heuristics FH1 to FH4 is shown in Table 4. Observe that after applying the four initial filtering heuristics, the set of selected conjectures has been narrowed to a total of four non-existence conjectures: one involving the concept *epv*, two for concept *epa*

Table 4. Iteration 2: Results of the application of filtering heuristics FH1-FH4.

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

Table 5. Iteration 2: Conjectures obtained after applying heuristics FH1-FH4.

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

and one involving the concept *idleF*. These conjectures are presented in Table 5. Note that these conjectures help overcome all the current failures, i.e. PO6, PO7, PO8, PO9, PO10 and PO11; therefore, the four conjectures are proposed as candidate invariants. Note also that when the new set of invariants are added to the model, two new failed POs are generated. These new failures are shown in Figure 22.

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

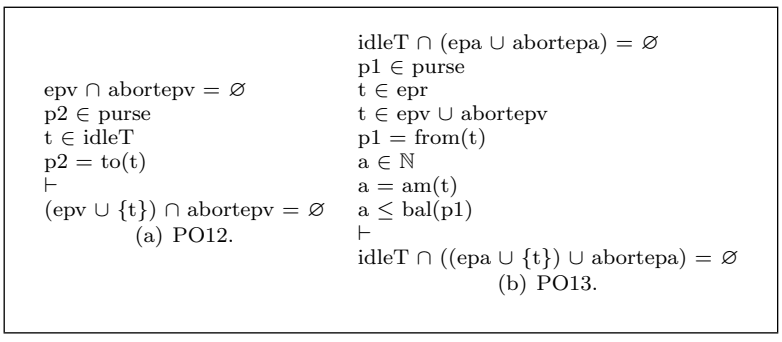
$$\begin{aligned} \textit{core concepts} &= \{\textit{epv, abortepv, idleT, epa, abortepa, purse,} \\ &\quad \textit{to, epr, from, am, bal}\} \\ \textit{non-core concepts} &= \{\textit{epv} \cap \textit{abortepv, epa} \cup \textit{abortepa,} \\ &\quad \textit{idleT} \cap \textit{(epa} \cup \textit{abortepa),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

and in the selection of the following PRs:

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

As with the first and second iterations, the filtering heuristics are applied in the first place to the core concepts identified in the goals of the failed POs. In this case, the search for invariants is focused in concepts *epv, abortepv, idleT, epa* and *abortepa*. The result of applying heuristics FH1-FH4 is shown in Table 6. Observe that the application of these filtering heuristics have limited the set of selected conjectures to a total of one non-existence conjecture involving the concept *epv*. This conjecture discharges both PO12 and PO13 and it does not generate any extra failed PO; thus, this conjecture is proposed as a candidate invariant. As no extra failures are generated with the introduction of this invariant, no further iterations are needed. The final set of invariants represented by the conjectures obtained from HRemo in the three iterations of our approach are shown in Figure 23.

The invariants shown in Figure 23 for level 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 8 invariants from the 17 used



(a) PO12. (b) PO13.

Fig. 22. Third set of failed POs.

Table 6. Iteration 3: Results of the application of filtering heuristics FH1-FH4.

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

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

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

$\text{idleT} \cap (\text{epv} \cup \text{abortepv}) = \varnothing$
(c) Third iteration.

Fig. 23. Mondex refinement fourth: automatically discovered invariants.

in [BY08], plus one invariant that implies two of the invariants suggested in [BY08]; that is, we identified invariant *(idleF ∪ epr) ⊆ idle* which implies invariants *idleF ⊆ idle* and *epr ⊆ idle* suggested in [BY08]. It is important to note that we have addressed all the failures produced in this refinement step. Some of the extra invariants used in [BY08] represent new requirements of the system, which are beyond the scope of our technique since we only target invariants needed to prove the refinement steps. However, some of these invariants are required in order to prove later refinements. This means that some extra failures would arise in the subsequent refinements. HRemo can be applied to explore these failures.

7.1.1. Overview of the application of HRemo to the full refinement chain of the Mondex case study

In the previous section we detailed the application of our approach to level 4 of the Mondex development. In Table 7 we summarise the results of applying the invariant discovery technique to each step of the refinement chain. As can be observed our technique succeeded in finding the invariants required to discharge all failed POs in levels 4, 5, 6 and 9. However, in levels 2, 3, 7 and 8 not all failures were discharged. Next, we explain for each level why the missing invariants were not identified by HRemo. Where possible, we estimate the length of time it would take to extend HR in order to enable it to identify the invariant in question.

The following invariants were not identified at level 2:

$$\forall\, p \cdot p \in purse \Rightarrow abal(p) = bal(p) + sum(am[pfrom[\{p\}]]) \tag{6}$$

$$\forall\, p \cdot p \in purse \Rightarrow lost(p) = sum(am[lfrom[\{p\}]]) \tag{7}$$

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

Table 7. Summary of results for the Mondex development.

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

represents a partial function which maps finite sets of natural numbers to natural numbers that represent the summation of each set, i.e:

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

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

sum(A,B)

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

The following invariants are the missing invariants from level 3:

$$pfrom^{-1} = (pending \lhd from) \tag{8}$$

$$lfrom^{-1} = (recover \lhd from) \tag{9}$$

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

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

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

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

$$\forall p \cdot p \in (active \setminus idleFP) \land p \in dom(currentF2) \Rightarrow currentF(p) = currentF2(p) \tag{10}$$

$$\forall p \cdot p \in (active \setminus idleTP) \land p \in dom(currentT2) \Rightarrow currentT(p) = currentT2(p) \tag{11}$$

$$\forall p, t \cdot p \in active \land from(t) = p \land currentSeqNo(p) = Fseqno(t) \Leftrightarrow currentF(p) = t \tag{12}$$

$$\forall p, t \cdot p \in active \land to(t) = p \land currentSeqNo(p) = Tseqno(t) \Leftrightarrow currentT(p) = t \tag{13}$$

All these are gluing invariants that describe how the abstract variables *currentF* and *currentT* are represented in the concrete level. Invariants (10) and (11) are not identified through HREMO because they conflict with heuristic FH2 since the left- and right-hand sides of the conjectures are not disjoint with respect to variables *currentF2* and *currentT2*. On the other hand, invariants (12) and (13) present the same problem that arose with invariants (8) and (9) at level 3. This again suggests that a new PR, that permutes the columns within a data table, would be worth investigating in the future.

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

$$\forall p, n \cdot (p \mapsto n \in used \Rightarrow n \leq currentSeqNo(p)) \tag{14}$$

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

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

Table 8. Summary of results for the Flash file development.

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

NA = Not Applicable

7.2. Flash file case study

This case study [Dam10] was motivated by a "verification grand challenge" posed by Holzmann [HJG08]. As mentioned previously, this development consists of two sub-models; one that handles the software component of the flash file system and one that handles the hardware component. In this section we show the application of HRemo to the hardware model. The refinement chain for this model consists of five steps:

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

Level 2: introduces the concept of a page register that is used as an intermediate storage when reading and writing data from/to a physical page of the flash disk.

Level 3: models the process of block reclamation which consists of erasing and reusing a set of pages from the flash disk.

Level 4: handles the relocation of pages from a block that is going to be used to a free block.

Level 5: refines the process of erasing a block from the flash disk.

Level 6: introduces the concept of a status register which is used to indicate if the flash device is ready and if the previous operation has succeeded or not.

Table 8 summarises the results of the invariant discovery process carried out by HRemo. Note that at levels 2 and 4, HRemo found a set of invariants that discharged all the failed POs. At level 3, however, not all the failures were discharged. Note also that when including only typing invariants at levels 5 and 6, the models did not generate any failed POs. This meant that HRemo was not required for the refinement steps modelled by levels 5 and 6.

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

$$\forall r \cdot r \in dom(trans\_func) \Rightarrow flash(r) = flash2(trans\_func(r)) \tag{15}$$

$$programmed\_pages2 = trans\_func[programmed\_pages] \tag{16}$$

$$dom(flash2) = programmed\_pages2 \tag{17}$$

These invariants describe the relationships between the physical and logical views of the content of pages in the flash device. Invariant (15) is invented by HR; however, HRemo does not choose it as a candidate invariant because it does not pass the filter imposed by heuristic FH2; i.e. the left- and right-hand sides of the invariants are not disjoint since variable *trans_func* appears on both sides. Invariants (16) and (17) are also invented by HR; however, both invariants are needed to discharge the same failure; therefore, they are not selected as candidate invariants since HRemo only works for failures that require no more than one invariant in order to be discharged.

7.3. Summary of results

Table 9 summarises the results of applying our approach to each of the Event-B models we used during the development and the evaluation stages. All the experiments were performed with only basic typing invariants included in the models, i.e. no gluing or system invariants were present in the models. Table 9 shows for each refinement step, the number of failed POs as well as the number of gluing and system invariants discovered through our approach. We also record the number of iterations involved in the invariant discovery process. Note that the rows marked with a cross symbol (✗) in the column *All failures addressed* show the refinement steps for which not all failures were addressed.

Table 9. Automatically discovered invariants.

Event-B model	Step	Failed POs	Invariants				All failures addressed	
			Automatically discovered					
			Glue	System	Total	Iteration		
Traffic light	Level 2	2	2	0	**2**	1	✓	Development set
Vending machine (Arith)	Level 2	6	3	0	**3**	1	✓	
Vending machine (sets)	Level 2	6	3	0	**3**	1	✓	
Cars on a bridge	Level 2	2	1	0	**1**	1	✓	
	Level 3	6	0	5	**5**	1	✓	
	Level 4	7	0	5	**5**	1	✓	
Location Access Controller	Level 2	0	NA	NA	**NA**	NA	✓	
	Level 3	2	0	2	**2**	1	✓	
	Level 4	3	0	2	**2**	1	✓	
		2	0	2	**2**	2	✓	
		3	0	3	**3**	3	✓	
	Level 5	3	1	2	**3**	1	✓	
		6	1	3	**4**	2	✓	
		8	0	6	**6**	3	✓	
		5	0	3	**3**	4	✓	
		4	0	3	**3**	5	✓	
		1	0	1	**1**	6	✓	
Mondex	Level 2	4	0	1	**1**	1	✓	Evaluation set
		3	0	0	**0**	0	✗	
	Level 3	3	0	0	**0**	1	✗	
	Level 4	5	5	0	**5**	1	✓	
		6	1	4	**5**	2	✓	
		1	0	1	**1**	3	✓	
	Level 5	3	0	3	**3**	1	✓	
		5	0	4	**4**	2	✓	
		4	0	2	**2**	3	✓	
	Level 6	15	5	1	**6**	1	✓	
		25	0	14	**14**	2	✓	
		14	0	8	**8**	3	✓	
		13	0	8	**8**	4	✓	
		10	0	5	**5**	5	✓	
		8	0	4	**4**	6	✓	
		6	0	3	**3**	7	✓	
		3	1	1	**2**	8	✓	
		3	1	0	**1**	9	✓	
	Level 7	15	2	0	**2**	1	✓	
		5	1	0	**1**	2	✓	
		5	0	0	**0**	3	✗	
	Level 8	2	0	0	**0**	1	✗	
	Level 9	14	10	0	**10**	1	✓	
Flash File System FL	Level 2	1	0	1	**1**	1	✓	
	Level 3	4	0	1	**1**	1	✗	
		3	0	2	**2**	2	✓	
		2	0	1	**1**	3	✓	
		4	0	1	**1**	4	✓	
		2	0	1	**1**	5	✓	
		1	0	0	**0**	6	✗	
	Level 4	4	0	3	**3**	1	✓	
		1	0	1	**1**	2	✓	
	Level 5	0	NA	NA	**NA**	NA	✓	
	Level 6	0	NA	NA	**NA**	NA	✓	

NA = Not Applicable

In Table 10 we compare our results with the actual invariants given in the literature for the models used in the development set. Note that the other developments are not compared because they were developed by us. Note also that the invariants associated with levels five and seven of the Mondex system are not given in the literature. While all automatically discovered invariants are subsets of the invariants given in the literature, it is important to highlight that the automatically discovered invariants were sufficient to prove most of the refinement steps we encountered during our experiments.

Although we have identified some limitations of our approach, as discussed in Sections 7.1.1 and 7.2, it can be observed from Tables 9 and 10 that the automatic discovery of invariants through HRemo has provided promising results:

Table 10. 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.

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

- In most cases, the set of invariants discovered through HRemo helped discharge all failed POs required to prove the refinement steps.
- The set of gluing invariants found by HRemo in each refinement step was almost identical to the set of gluing invariants given in the literature – remember that the exception cases in the Mondex model, i.e. levels 2, 3 and 8, cannot be handled by HR's PRs – which have not been tailored for formal modelling.

With respect to system invariants, it can be observed that the last refinement of the cars on a bridge system shows a big gap between the invariants given in the literature and those found automatically with HRemo. As mentioned previously, we believe that this difference can be explained by the introduction of new requirements, resulting in the need for extra properties within the model.

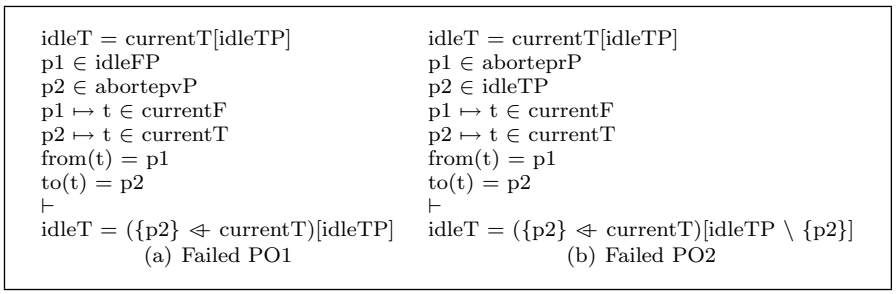
Moreover, HRemo has shown that it works better at generating small invariants rather than compound invariants. This is demonstrated at levels 5 and 6 of the location access controller and within the Mondex case study, where HRemo identified more system invariants than were given in the literature. Both refinement steps introduce a partition of sets. In Event-B a partition can be specified by a single invariant using the predicate *partition(s, s_1, ... s_n)*, where s_1, ... s_n partition the set s, or by multiple invariants defining the containment and disjointness relationships, i.e. $s = s_1 \cup ... \cup s_n$ and $s_i \cap s_j = \varnothing$. HRemo generated invariants of the second type; this is reflected in the number of automatically discovered invariants being greater than the number of invariants given in the literature.

7.4. Discussion: Beyond current applications

Debugging models: The invariant discovery process performed by HRemo helped us discover an inconsistency between levels 5 and 6 of our re-constructed Mondex system model. In the last iteration of invariant discovery, HRemo fails to find conjectures to discharge the two failed POs. At this point the invariant discovery process is no longer applicable since no failures are addressed by the iteration. This may occur because:

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

The failed POs generated by the model at this iteration are shown in Figure 24. These POs represent failures in the preservation of one of the invariants discovered by HRemo, that is:



(a) Failed PO1 (b) Failed PO2

Fig. 24. Level 6 Iteration 10: Failed POs.

$$idleT = currentT[idleTP]$$

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

The failed POs shown in Figure 24 are in fact unprovable. As can be observed, the equality expressed by the invariant holds in the hypothesis of the POs; however, the right-hand side of the equality is modified in the goals by removing the element *p2* from the domain of function *currentT* while the left-hand side remains unmodified. The events associated to the failed POs are new events introduced in the concrete model. Their role is to register the end of a transaction when one of its sides has failed while the other is still in the idle state. The failed POs suggest that:

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

In order to solve these failures either an abstract version of the events should be modelled in level 5 of the development, or the invariant at level 6 should be removed or modified. By analysing the model we realised that in level 5 we had not handled all possible cases for which a transaction could end because of an abort state. The inconsistency was solved by adding new events in level 5 that handled all possible states in which a transaction could fail.

Note that this analysis is not currently performed by HRemo. However, through the automatically discovered invariants we found the inconsistency in the model. These failures are not generated in the original development of the Mondex system presented in [BY08], because the gluing invariant is defined as:

$$currentT[idleTP] \subseteq idleT$$

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

Handling incorrect models: As mentioned before, our approach to invariant discovery assumes that the models are correct. Having the ability of identifying incorrect models would increase the effectiveness of HRemo. We have developed a technique that aims at providing modelling guidance when a failed refinement is closely aligned with a known pattern, we call this *refinement plans* [GIL12]. Refinement plans are used to identify if a known pattern of refinement is closely aligned to the given failed refinement. Each refinement pattern is associated with a set of critics – where a critic represents a common pattern of failure at the level of POs and models. Moreover, associated with each critic is generic modelling guidance as to how to overcome the failure, e.g. the speculation of missing invariants and events, etc. Refinement plans complement HRemo by identifying incorrect models and providing guidance in overcoming failures.

Furthermore, when a common pattern of failure is instantiated by a particular refinement step, the associated guidance will typically only be partially instantiated. This is the case when the guidance includes an invariant schema. In such cases HRemo can be used to complete the instantiation of a partially instantiated invariant schema.

Applying the approach to other formalisms: As mentioned previously, we believe that our approach could be applied more widely. In order to illustrate this we show how the data from a simple model of a vending machine written in the Z notation maps to the aspects required in our approach. The example, which is taken from [WD96], is shown in Figure 25. The specification models a vending machine which dispenses

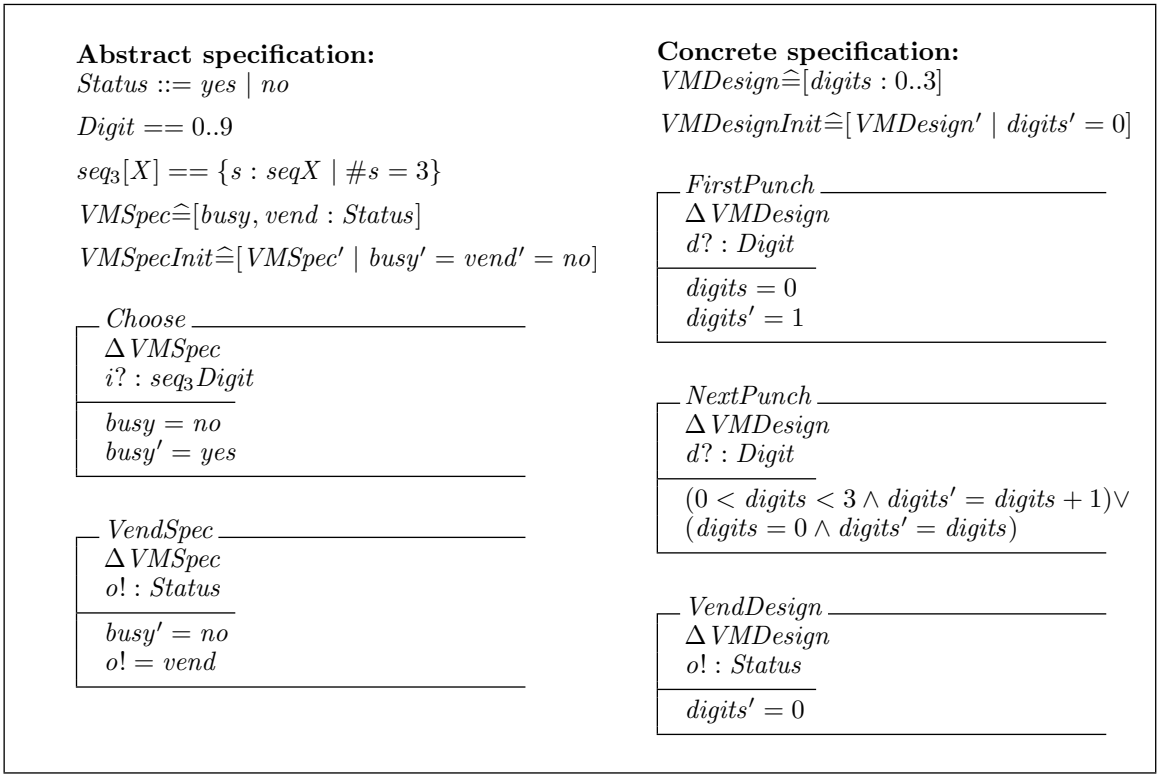


Fig. 25. Z specification of a vending machine system taken from [WD96]

drinks when a three-digit code is typed by a customer. In the abstract model, the action of entering the code to the machine is modelled by an atomic step, i.e. the operation *Choose*, while in the concrete model the three digits of the code are input one-by-one, i.e. operations *FirstPunch* and *NextPunch*.

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

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

where the examples of *Status* and *Digit* are {*yes*, *no*} and {0,1,2,3,4,5,6,7,8,9}, respectively. The examples required for the T2 concepts can be obtained from an animator for Z; for instance, the ProZ animator [PL07]. In order to prove that the concrete schema *FirstPunch* refines the abstract schema *Choose*, the following PO must be proved:

$$\begin{aligned} &\forall\, \textit{busy}, \textit{vend} : \textit{Status};\ \textit{digits}, \textit{digits}' : 0..3;\ \textit{seq}_3\textit{Digit};\ d? : \textit{Digit} \bullet \\ &\quad \textit{busy} = \textit{no} \land \textit{digits} = 0 \land \textit{digits}' = 1 \Rightarrow \exists\, \textit{busy}', \textit{vend}' : \textit{Status} \bullet \textit{busy}' = \textit{yes} \end{aligned}$$

As it stands, the PO is not provable; a retrieve relation that explains the correspondence between the abstract and concrete models is required. HRemo can be used to find the retrieve relation. In order to do so, the non-core concepts must be identified from the failed PO, i.e.

$$\{\textit{busy=no, digits=0, digits=1, busy=yes, (busy = no} \land \textit{digits = 0} \land \textit{digits = 1)}\}$$

The first four non-core concepts can be represented in HRemo by applying the *split* PR to core concepts

busy and *digits* where the split values are *no, 0, 1* and *yes*. The remaining non-core concept can be obtained through the application of the *compose* PR. The required retrieve relation is:

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

We have previously shown that HRemo is capable of identifying this kind of invariant.

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

8. Related 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.

8.1. 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, 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[8], 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.

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 inference engines 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.

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

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.

8.1.1. 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. Moreover, two alternative Java representations were generated in both experiments in order to illustrate one of the characteristics of Daikon –that the invariant discovery process is influenced by the syntax of the code. We come back to this later in this section.

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 26 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 26 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 11 shows the results of the invariant analysis from both approaches for the models shown in Figure 26. 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

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

First Java 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;
   }
  }
}
```

Second Java 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;
  }
}
```

Fig. 26. Event-B and Java models of the cars on a bridge system.

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 11, 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 experiment are shown in Table 12. 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

Table 11. Comparison of expected and detected invariants for the first refinement of the "Cars on a bridge" model.

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 ...

colour of the traffic lights, i.e. *cob.Color.RED == this.il_tl* and *cob.Color.GREEN == this.ml_tl* and the number of cars going from the island to the mainland, i.e. *c == 0*; nevertheless Daikon does not produce global relationships between these values; for instance that *ml_tl=green* $\Rightarrow$ *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 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 low 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

Table 12. Comparison of expected and detected invariants for the second refinement of the "Cars on a bridge" model.

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

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.

8.2. 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 ATF 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.

9. Conclusions and Further work

Building upon HR, animation and proof-failure analysis, we have presented HREMO – an automated approach to invariant discovery. The key contribution of our work is the set of heuristics which 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. 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.

As noted in Section 3.2, 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 that increasing the randomness in the production of the traces is required in order to improve the possibility of generating the missing invariants.

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, as mentioned in Section 7.4, we have developed *Refinement Plans* to handle failed refinement steps and we use the instantiated patterns to further tailor the search in HREMO. Furthermore, 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.

We have identified two areas in which HREMO can be tailored for the formal modelling context:

1. reducing the number of conjectures generated by HR, and
2. adding new PRs suitable within formal modelling.

We have carried out initial experiments to address 1. A reduction in the number of generated conjectures can be achieved by constraining the concepts allowed within the theory to only concepts whose set of variables are disjoint. In the first iteration of the Mondex development presented in Section 7.1, 7296 conjectures were generated; by applying only this constraint, 742 conjectures were created instead. Regarding 2, we have highlighted in Section 7 the need for new PRs. Specifically a PR that allows the permutation of columns within a data table. We have also identified limitations in the application of current PRs, e.g. the *numRelation* PR. Addressing these limitations, and the development of new PRs, are part of our future work agenda.

Currently most of the invariant discovery process within HREMO has been automated; however, heuristics FH4 and FH5 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.

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].

Finally, in [BS10] the authors discuss an interesting technique where invariants are calculated for protocols by identifying relationships between the states of the running protocol at different levels of abstraction. We would like to investigate cases in which this approach fails, with the aim of determining whether HREMO could address the failures.

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. We also thank the reviewers of the 2011 Refinement Workshop and the reviewers of the Journal of Formal Aspects of Computing for their valuable comments. The research reported in this paper is supported by EPSRC grants EP/F037058, EP/F035594 and EP/J001058. Maria Teresa Llano was also supported by a BAE systems studentship.

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.

[Baa88] B. Baars. *A cognitive theory of consciousness*. Cambridge University Press, 1988.

[Baa97] B. Baars. *In the theater of consciousness: The workspace of the mind*. Oxford University Press, 1997.

[BGA67] J. Bruner, J.J. Goodnow, and G.A. Austin. *A study of thinking*. Science Editions, New York, 1967.

[Bol05] C. Bolton. Using the Alloy analyzer to verify data refinement in Z. *Electronic Notes in Theoretical Computer Science*, 137(2):23–44, 2005.

[BS10] R. Banach and G. Schellhorn. Atomic actions, and their refinements to isolated protocols. *Formal Aspects of Computing*, 22(1):33–61, 2010.

[Buc75] B. Buchanan. Applications of artificial intelligence to scientific reasoning. In *Second USA-Japan Computer Conference*, pages 189–194, Tokyo, 1975. AFIPS and IPS I.

[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.

[CBW99] S. Colton, A. Bundy, and T. Walsh. Automatic concept formation in pure mathematics. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 786–793, 1999.

[CBW00a] S. Colton, A. Bundy, and T. Walsh. Automatic identification of mathematical concepts. In *Proceedings of the 17th International Conference on Machine Learning*, pages 183–190. Morgan Kaufmann, 2000.

[CBW00b] S. Colton, A. Bundy, and T. Walsh. Automatic invention of integer sequences. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pages 558–563, 2000.

[CBW00c] S. Colton, A. Bundy, and T. Walsh. On the notion of interestingness in automated mathematical discovery. *International Journal of Human Computer Studies*, 53(3):351–375, 2000.

[CC08] J. Charnley and S. Colton. A global workspace framework for combining reasoning systems. In *Proceedings of the Symposium on the Integration of Symbolic Computation and Mechanised Reasoning*, pages 261–265, 2008.

[CCM06] J. Charnley, S. Colton, and I. Miguel. Automatic generation of implied constraints. In *Proceedings of the 17th European Conference on AI*, pages 73–77, 2006.

[Cha10] J. Charnley. *A Global Workspace Framework for Combined Reasoning*. PhD thesis, Imperial College, London, 2010.

[CM01] S. Colton and I. Miguel. Constraint generation via automated theory formation. In *7th International Conference on the Principles and Practice of Constraint Programming*, pages 575–579, 2001.

[CM06] S. Colton and S. Muggleton. Mathematical applications of Inductive Logic Programming. *Machine Learning*, 64:25–64, 2006.

[Col99] S. Colton. Refactorable numbers - a machine invention. *Journal of Integer Sequences*, 2, 1999.

[Col02a] S. Colton. *Automated Theory Formation in Pure Mathematics*. Springer-Verlag, 2002.

[Col02b] S. Colton. The HR program for theorem generation. In *CADE'18*, volume 2392 of *Lecture Notes in Computer Science*, pages 37–61. Springer, 2002.

[CP04] S. Colton and A. Pease. The TM system for repairing non-theorems. In *Workshop on Disproving, Proceedings of IJCAR'04*, pages 13–26, 2004.

[CP05] S. Colton and A. Pease. The TM system for repairing non-theorems. In *Selected papers from IJCAR'04 disproving workshop, Electronic Notes in Theoretical Computer Science*, volume 125(3), pages 87–101. Elsevier, 2005.

[CS02] S. Colton and G. Sutcliffe. Automatic generation of benchmark problems for automated theorem proving systems. In *Proceedings of the 7th AI and Maths Symposium*, 2002.

[Dam10] K. Damchoom. *An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B*. PhD thesis, University of Southampton, 2010.

[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, 2007.

[GIL12] G. Grov, A. Ireland, and M.T. Llano. Refinement plans for informed formal design. In *ABZ*, Lecture Notes in Computer Science, pages 208–222. Springer, 2012.

[HJG08] G.J. Holzmann, R. Joshi, and A. Groce. 25 years of model checking. chapter New Challenges in Model Checking, pages 65–76. Springer-Verlag, 2008.

[IEC+06] A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[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.

[Lak76] I. Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.

[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.

[Len77] D. Lenat. Automated theory formation in mathematics. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 833–842. Morgan Kaufmann, 1977.

[LIP11] M.T. Llano, A. Ireland, and A. Pease. Discovery of invariants through automated theory formation. In *Proceedings of the 15th International Refinement Workshop*, volume 55 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–19. Open Publishing Association, 2011.

[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.

[McC94a] W. McCune. A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, Argonne National Laboratories, 1994.

[McC94b] W.W. McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, USA, 1994.

[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.

[MIG11] E. Maclean, A. Ireland, and G. Grov. The CORE system: Animation and functional correctness of pointer programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2011): Tool Demonstration Paper*, pages 588–591, Lawrence, Kansas., 2011. IEEE.

[MRMDB11] O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, 39:1637–1646, 2011.

[MSC02] A. Meier, V. Sorge, and S. Colton. Employing theory formation to guide proof planning. In *AISC/Calculemus'02, LNAI 2385*, pages 275–289. Springer, 2002.

[PCCng] A. Pease, S. Colton, and J. Charnley. Automated theory formation: The next generation. *IFCoLog Lectures in Computational Logic*, 2012 (Forthcoming).

[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.

[PL07] D. Plagge and M. Leuschel. Validating Z specifications using the ProB animator and model checker. In *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer-Verlag, 2007.

[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, 2010.

[RH90] G. Ritchie and F. Hanna. AM: a case study in methodology. In *The foundations of AI: a sourcebook*, pages 247–265. Cambridge University Press, 1990.

[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.

[SCMM08] V. Sorge, S. Colton, R. McCasland, and A. Meier. Classification results in quasigroup and loop theory via a combination of automated reasoning tools. *Comment.Math.Univ.Carolin*, 49(2):319–339, 2008.

[SMMC08] V. Sorge, A. Meier, R. McCasland, and S. Colton. Automatic construction and verification of isotopy invariants. *Journal of Automated Reasoning*, 40(2-3):221–243, 2008.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice-Hall, 1996.

[Win70] P. Winston. Learning structural descriptions from examples. Technical Report TR-231, MIT, 1970.

[ZFCS02] J. Zimmer, A. Franke, S. Colton, and G. Sutcliffe. Integrating HR and tptp2X into MathWeb to compare automated theorem provers. In *Proceedings of the CADE'02 Workshop on Problems and Problem sets*, 2002.