Proving Event-B models with reusable generic lemmas

Alexei Iliasov, Paulius Stankaitis, Alexander Romanovsky

Newcastle University, UK

Abstract. Event-B is one of more popular notations for model-based, proof-driven specification. It offers a fairly high-level mathematical language based on FOL and ZF set theory and an economical yet expressive modelling notation. Model correctness is established by proving a number of conjectures constructed via a syntactic instantiation of schematic conditions. A significant part of provable conjectures requires proof hints from a user. For larger models this becomes extremely onerous as identical or similar proofs have to be repeated over and over, especially after model refactoring stages. In the paper we discuss an approach to making proofs more generic and thus less fragile and more reusable. The crux of the technique is offering an engineer an opportunity to complete a proof by positing and proving a generic lemma that may be reused in the same or even another project. To assess the technique potential we have developed a plug-in to Rodin Platform and used it to proof a number of pre-existing Event-B models.

1 Introduction

There was a concerted effort, funded by a succession of EU research projects [12,19], to make Event-B [4] and its toolkit, Rodin Platform [23], appealing and competitive in an industrial setting. One of the lessons of this mainly positive exercise is the general aversion of industrial users to interactive proof. It is possible, in principle, to learn, through experience and determination, the ways of underlying verification tools and master refinement and decomposition to minimize proof effort. The methodological implications are far more serious: building a good model is necessarily a trial and error process; one often has to start from a scratch or do considerable refactoring to produce an adequate model. This, obviously, necessitates redoing proofs and makes time spent proving dead-end efforts seem pointlessly wasted. Hence, proof-shy engineers too often do not make a good use of formal specification stage as they tend to hold on to the very first, often incoherent design.

We want to change the way proofs are done, at least in an industrial setting. In place of an interactive proof - something that is inherently a one-off effort in Event-B and comparable model-based notations - we incite modellers to gardually accumulate a library of general support condition called a *schematic lemmas*. The principle here is that a fitting schematic lemma added to hypothesis set would discharge an open proof obligation. Such a lemma may not refer to any model variables or user-defined types and is, in essence, a property supporting the definition of the underlying mathematical language [1]. From our experience, a modelling project has a fairly distinctive usage of mathematical language and, we hypothesise, this leads to a distinctive set of supporting lemmas.

Since a schematic lemma does not reference model-specific variables or types it can be immediately reused in a new context and thus is a tangible and persistent outcome of a modelling effort, even an abortive one. It is not affected by model refactoring and restructuring of refinement steps. In a longer term, we see schematic lemmas as a methodological tool promoting winder application of model restructuring (or even restarting from scratch) and thus helping engineers to construct better models and not feel constrained by the cost of a proof effort.

Another intriguing possibility, yet untested in practice, is that for a narrow application domain combined with tailored development patterns it is feasible to reach a point where schematic lemma library makes modelling nearly free of interactive proofs.

The rest of the paper is organised as follows. In Section 2 we briefly present the Event-B modelling notation as well as its verifications rules; we also introduce the Why3 plug-in that makes use of the Why3 umbrella prover [9]. Section 3 expands on the idea behind schematic lemmas and their potential role as a proof process. We present some experimental results in Section 4 and summarise the findings in Section 5.

2 Background

2.1 Event-B

We apply the Event-B [4] formal modelling notation to specify and verify railway signalling. Event-B belongs to a family of state-based modelling languages that represent a design as a combination of state (a vector of variables) and state transformations (computations updating variables).

In general, a design in Event-B is abstract: it relies on data types and state transformations that are not directly realisable. This permits terse models abstracting away from insignificant details and enables one to capture various phenomena of a system with a varying degree of detail. Each statement about the effect of a certain computation is supported by a formal proof. In Event-B, one is able to make statements about safety (this incorporates the property of functional correctness) and progress. Safety properties ensure that a system never arrives at a state that is deemed unsafe (i.e., a shaft door is never open when a lift cab is on a different floor). Progress properties ensure that a system is able to achieve its operational goals (i.e., a lift cab eventually arrives).

Being a general-purpose formalism, Event-B does not attempt to fit any specific application domain. It has found applications in hardware modelling,

[1] There are, however, cases where where the modeller's insight is critical in providing a witness or case split. These, we believe, should be handled at the specification as discussed, for instance, in [11].

```
machine M
  sees Context
  variables $v$
  invariant $I(c, s, v)$
  initialisation $R(c, s, v')$
  events
    $\mathsf{E}_1$ = any $vl$ where $g(c, s, vl, v)$ then $S(c, s, vl, v, v')$ end
    ...
end
```

Fig. 1. Event-B machine structure.

validation of high-level use case scenarios, verification of business process logics and even as a friendly notation for a mathematician looking for a support from machine provers.

An Event-B development starts with the creation of a very abstract specification. A cornerstone of the Event-B method is the stepwise development that facilitates a gradual design of a system implementation through a number of correctness-preserving *refinement* steps. The general form of an Event-B model (or *machine*) is shown in Fig. 1. Such a model encapsulates a local state (program variables) and provides operations on the state. The actions (called *events*) are characterised by a list of local variables (parameters) vl, a state predicate g called *event guard*, and a next-state relation S called *substitution* or event *action*.

Event parameters and guards may be omitted leading to syntactic short-cuts starting with keywords `when` and `begin`.

Event guard g defines the condition when an event is *enabled*. Relation S is given as a generalised substitution statement [3] and is either deterministic ($x := 2$) or non-deterministic update of model variables. The latter kind comes in two notations: selection of a value from a set, written as $x :\in \{2, 3\}$; and a relational constraint on the next state v', e.g., $x :| \; x' \in \{2, 3\}$.

The `invariant` clause contains the properties of the system, expressed as state predicates, that must be preserved during system execution. These define the *safe states* of a system. In order for a model to be consistent, invariant preservation is formally demonstrated. Data types, constants and relevant axioms are defined in a separate component called *context*.

Model correctness is demonstrated by generating and discharging *proof obligations* - theorems in the first-order logic. There are proof obligations for model consistency and for a refinement link - the forward simulation relation - between the pair of *abstract* and *concrete* models.

More details on Event-B, its semantics, method and applications may be found in [4] and also on the Event-B community website [8]. A concise discussion of the Event-B proof obligations is given in [10].

2.2 Why3 plugin

It has been long recognised that Rodin Platform may significantly benefit from an interface between Event-B and TPTP [24] provers. To simplify translation we decided to use Why3 [9] umbrella prover that offers a single and quite palatable input notation and also supports SMT-LIB compliant provers. Why3 supports 16 external automatic provers (not counting different versions of the same tool) and these include all the state-of-the-art tools like Z3 [14], SPASS [5], Vampire [16] and Alt-Ergo [20].

Given that provers are CPU and memory intensive and there is a great potential for exploiting parallel processing, from the outset we were aiming at provers-as-a-service cloud architecture. Indeed, running a collection of (distinct) provers on the same conjecture is a trivial and fairly effective way to speed up proofs given plentiful resources. Usability perception of interactive modelling methods such as Event-B is sensitive to peak performance when a burst of activity (new invariant) is followed by a relatively long period of idling (modeller thinking and entering model). The cloud paradigm, where only the actual CPU time is rented, seems well suited to such scenario. Also, the cloud's feature of scalability plays a critical role in this situation.

A plug-in to Rodin Platform was realised **??** to map between the Event-B mathematical language and Why3 *theory* input notation (we do not make use of its other part - a modelling language notation). The syntactic part of the translation is trivial: just one Tom/Java class mapping between Event-B and Why3 operators. The bulk of the effort is in the axioms and lemmas defining the properties of the numerous Event-B set-theoretic constructs. We have a working prototype able to discharge (via provers like SPASS and Alt-Ergo) a number of properties that previously required interactive proof. At the same time, we realize that axiomatisation of a complex mathematical language like the one of Event-B is likely to be an ever open problem. It is apparent that different provers prefer differing styles of operator definitions: some perform better with an inductive style (i.e., to define set cardinality one may say that the size of an empty set is zero, adding one element to a set increases its size by one) while others prefer regress to already known concepts (there exists a bijection such that ...). Since we do not know how to define one best axiomatization, even for any one given prover, we offer an open translator with which a user may define, with as many cross-checks as practically reasonable, a custom embedding of Event-B into Why3.

The Why3 theory library we have developed in the support of the axiomatisation of the Event-B mathematical language does not appear optimal yet. For most cases the Why3 plug-in performs on par or better than SMT plug-in [7] although it takes longer while using more provers at the back-end. With one model (of a train control system), we had a disappointing result of 32 undischarged proof obligations with Why3 plug-in against 5 left undischarged by the SMT plug-in.

3 Schematic lemmas

There is a number of circumstances when existing interactive proofs become invalidated and a new version of an undischarged proof obligation appears.

On rare occasions a model or its sizeable part are changed significantly so that there is no or little connection between old and new proof obligations. Far more common are incremental changes that alter the goal, set of hypotheses, identifier names or types. During the refactoring of a refinement tree it is very common to lose a large proportion of manual proofs.

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

Even in the case of a significant model change, it is, in our experience, likely that proof obligations similar to those requiring an interactive proof re-appear. In addition, there is a large number of essentially identical interactive proofs re-appearing in different projects due to specific weaknesses in the underlying automatic provers.

The key to our approach is understanding what 'similar' means in the relation to some two proof obligations. One interpretation is that similar conditions can be discharged by the same proof scripts. To make it practical, this has to be relaxed with some form of a proof script template [15]. The interpretation we take in this work is that two proof obligations are similar if they both can be discharged by adding same schematic lemma to the set of their hypotheses. This definition is rather intricately linked with the capabilities of underlying automated provers: adding a tautology (a proven lemma) to hypotheses does not change a conjecture but it might help to guide an automated prover to successful proof completion.

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

There are situations when the only viable way to complete a proof is by providing a proof hint. One such case - refinement of event parameters - is adequately addressed at the modelling notation level where a user is requested to provide a witness as a part of a specification. There are proposals to generalise this, for the majority of situations, and define hints at the model level [11].

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

$$\begin{array}{l} library \in \mathrm{BOOKS} \rightarrow \mathbb{N} \\ b \in \mathrm{BOOKS} \wedge c \in \mathbb{N} \\ \ldots \\ \vdash \\ library \mathbin{\lhd\!\!+} \{b \mapsto c\} \in \mathrm{BOOKS} \rightarrow \mathbb{N} \end{array}$$

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

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

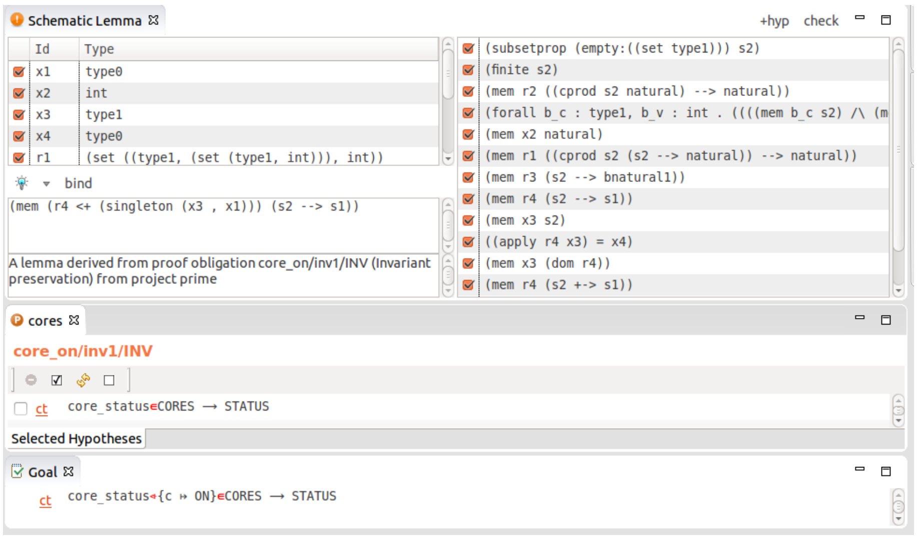
Since the Event-B mathematical language does not have type variables such a condition may only be defined either for specific A's and B's, or, in a slightly altered form, using the Theory plug-in [17]. But to discharge the original proof obligation one still needs to find this lemma and instantiates it. It is a tedious and error-prone process for a human but a fairly trivial task for a certain kind of automated provers.

The example above is quite generic in the sense it is potentially useful for in many other contexts. At times a schematic lemma need to be fairly concrete (see examples in Section 4. It is also easier to write a lemma that narrowly targets a proof obligation. This distinction between 'general' and 'specific' is, at the moment, completely subjective and relies on the modeller's intuition. To reflect the fact that a more general lemma is more likely to be reused, schematic lemmas are classified into three visibility classes: machine (single model), project (collection of models) and global. A machine-level lemma will be considered for a proof obligation of the machine with which the lemma is associated; similarly, for the project-level attachment. A global schematic lemma becomes a part of the Event-B mathematical language definition for the Why3 plug-in.

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

3.1 Automatically including relevant lemmas

Once there is a library of lemmas in place, it is vital that there is a way to automatically may use of them in every new proof obligation. Including all the



	Id	Type
✓	x1	type0
✓	x2	int
✓	x3	type1
✓	x4	type0
✓	r1	(set ((type1, (set (type1, int))), int))

```
(mem (r4 <+ (singleton (x3 , x1))) (s2 --> s1))
```

A lemma derived from proof obligation core_on/inv1/INV (Invariant preservation) from project prime

```
(subsetprop (empty:((set type1))) s2)
(finite s2)
(mem r2 ((cprod s2 natural) --> natural))
(forall b_c : type1, b_v : int . ((((mem b_c s2) /\ (m
(mem x2 natural)
(mem r1 ((cprod s2 (s2 --> natural)) --> natural))
(mem r3 (s2 --> bnatural1))
(mem r4 (s2 --> s1))
(mem x3 s2)
((apply r4 x3) = x4)
(mem x3 (dom r4))
(mem r4 (s2 +-> s1))
```

core_on/inv1/INV

```
ct   core_status∈CORES ⟶ STATUS
```

```
ct   core_status⩤{c ↦ ON}∈CORES ⟶ STATUS
```

Fig. 2. Schematic lemma prover interface. Instead of working with the built-in interactive prover, a modeller attempts to construct a provable schematic lemma that would discharge the current proof obligation.

lemmas in the hypotheses of every conjecture would simply overwhelm provers and effectively preclude automated proof. To discover relevant schematic lemmas we match the structure of a lemma against the structure of conjecture goals and hypotheses. Recall that a schematic lemma has no free identifiers and thus matching must be over structure.

Directly comparing a lemma and a conjecture is expensive: a straightforward algorithm (tree matching) is quadratic unless memory is not an issue. We use a computationally cheap proxy measure known as the Jaccard similarity which, as the first approximation, is defined as $JS(P, Q) = \text{card}(P \cap Q)/\,\text{card}(P \cup Q)$.

The key is in computing the number of overall and common elements and, in fact, defining what an ”element” means for a formula. One immediate issue is that P and Q are sets and a formula, at a syntactic level, is a tree. One common way to match some two sequences (e.g., bits of text) using the Jaccard similarity is to use *shingles* of elements to attempt to capture some part of the ordering information. A shingle is a tuple preserving order of original elements but seen as an atomic element. Thus sequence $[a, b, c, d]$ could be characterised by two 3-shingles $P = \{[a, b, c], [b, c, d]\}$ (here $[b, c, d]$ is just a name) and matching based on these shingles would correctly show that $[a, b, c, d]$ is much closer to $[a, b, c, d, e]$ than to $[d, c, b, a]$. To account for trees structure we do matching on a set of paths from a root to all leaves and also on the the set of sequences of the form $[p, c_1, \ldots, c_2]$ where p is a parent element and $c_1, \ldots, c_2$ are children. This immediately gives a set of n-shingles that might need to be converted into shorter m-shingles to make things practical.

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

$$s(P, Q) = \sum_{i \in I_1} \mathrm{w_d}(i) + c \sum_{i \in I_2} \mathrm{w_w}(i) \qquad I_1 = \mathrm{sd}(P) \cap \mathrm{sd}(Q), I_2 = \mathrm{sw}(P) \cap \mathrm{sw}(Q)$$

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

3.2 Schematic lemma plug-in

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

There are three main parts to the definition of a schematic lemma: identifiers, hypotheses and the goal. Identifier definition may use either one of the two built-in types (boolean and integer) or a fresh type variable (i.e., *type*0 in Fig. 2). Hypotheses are defined by a list of predicates (while logically order should not matter, in practice it does and it is advantageous to have more constricting hypotheses first); these predicates may not mention any model variables but can refer to the identifiers defined in the lemma. And the goal is a predicate over the lemma identifiers.

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

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

To aid in the construction of a schematic lemma, the plug-in provides some simple productivity mechanisms. A hypotheses can be deselected without removing it to check whether both the lemma goal and the context proof obligation are still provable. An identifier may also be deselected and this automatically deselects all the hypotheses mentioning the identifier. It will take more experiments to arrive at methodological guidelines on constructing lemmas.

4 Case Study

In this section we discuss the experience of applying the schematic lemmas technique to prove several pre-existing models. Since this is an on-going project, we also discuss perceived advantages and disadvantages of doing proofs with our technique.

As the case study we consider four models, some of them fairly well known to the Event-B community. They are not very large but still have a reasonable number of proof obligations and make a good use of refinement and Event-B modelling notation. Our intention was to take models from different domains constructed by different people to see how the technique performs in different settings. On the whole we were pleased to find that such diverse models still share a lot of schematic lemmas and it supports our conjecture that it is worthwhile to build lemma library. We do not have enough to show that this process definitely leads to a saturation point but we did observe that each subsequent model we tackled was a little bit easier since lemmas are reused.

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

4.1 Automatic Proving

The core of the experiment was to apply the schematic lemma plug-in to several diverse models and compare results with the existing proof infrastructure including the Why3 plug-in not equipped with schematic lemmas. The Rodin Platform provides facility to define automatic tactics, combining certain rewrite rule and automatic provers, and apply them redo all the proofs of a project. For this experiment, we have defined four such tactics and compared their performance. We have made every attempt to make best use of the available tools such as Atelier-B ML prover, built-in PP and nPP provers, and, of course, the SMT plug-in that relies on on some of the same back-end SMT provers.

Table 1 summarises the results of our experiment. We use two tactics that are commonly available to Rodin users. Tactic^1 applies a number of rewrite rules and then tries nPP, PP and ML provers; Tactic^2 does the same with addition of the SMT plug-in. The Why3 tactic is similar to Tactic^1 but with Why3 plug-in as the sole automatic prover. This tactic does not use any schematic lemmas and relies solely on the basic axiomatisation library defining various Event-B operators. In

Model	Proof obligations	open, Tactic[1]	open, Tactic[2]	open, Why3	open, Why3 (+ SL)
Order/Supply Communication [1]	276	24	4	8	4 (+2)
Fisher's Algorithm [2]	82	16	4	1	0 (+1)
Train Control System [4] (Chapter 17)	133	36	5	32	32 (+0)
B2B Communication prot. [22]	498	63	25	20	8 (+5)
Automated Teller Machine [18]	962	77	28	1	0 (+1)
Total	1951	216	66	62	12

Table 1. Comparative performance of four proof tactics; the first column is the overall number of generated proof obligations, the following four columns give the number of proof obligations remaining open (undischarged) after applying, from a scratch (that is, purging any previous proofs) the certain proof tactic. The final column gives in brackets the number of schematic lemmas used in the model (but not necessarily defined specifically for the model).

the last column, the Why3 plug-in is able to locate an include suitable schematic lemmas. This is a completely automatic process: one can define a number of schematic lemmas (when doing interactive proofs), then purge all the proofs and the lemmas will be picked up automatically when relevant. The last number (+x) is the number of used schematic lemmas.

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

4.2 Nesting lemmas

In this subsection we go a bit a deeper and discuss one specific example where a schematic lemma is used to complete a proof. We approached the experiment in a more or less blind style where a model itself was not analysed in any detail and we were generally concerned only with the specifics of a proof obligation - its goal and hypothesis, - in an attempt to deduce a schematic lemma strong enough to discharge the condition.

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

The solution is to allow a modeller to construct chains of lemmas of which only the last one is used in the capacity of a schematic lemma and the rest help to prove it. With extra support lemmas one should be able to handle pretty much any case of forward or backward proof. These additional lemmas are visible in the context and saved with the schematic lemma so that one is able to redo all the proofs strictly on the basis of Why3 axiomatisation library. Another possibility, offered by Why3 itself, is to transition to a far more capable environment of Isabelle or Coq and complete a proof there. We have not tried this route so far and it is not clear how to embed an external proof script in a schematic lemma.

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

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

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

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

It seems to be a promising start as the original proof obligation was now discharged by Alt-Ergo (among others) in just 0.03s. Yet the lemma itself could not be proven.

We discovered two new lemmas that should be added in the context of the schematic lemma and are enough to discharge it. They state some simple properties about domain overriding, and the functionality of an overridden function.

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

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

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

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

Table 2. The dynamics of proving the B2B Communication protocol model using the schematic lemma technique. The numbers show how each next lemma (L_1, L_2, ...) affects the overall number of open proof obligations.

4.3 Lemma reuse

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

In this experiment, we address the problem of proof re-usability by shifting the focus from proving a single verification condition to validating remaining undischarged proof obligations of the model. We use a publicly available model Buyer/Seller B2B Communication protocol [22]. In our view, it is a fairly typical example of a model not constructed solely for illustration purposes, i.e., there is some scale and purpose to it.

A Buyer/Seller B2B Communication protocol model has 11 refinement steps and 498 verification conditions. Combining all the default tactics with all the available automatic provers and the SMT plug-in results in 25 undischarged verification conditions (63 without the SMT plug-in).

Our standard routine based on the Why3 plug-in consists in first applying the plug-in without any schematic lemmas with increasingly longer timeouts and only afterwards reviewing remaining conditions for the purpose of writing schematic lemmas.

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

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

To pick but few simpler examples we show again a variation of reasoning about functional override and finiteness. Some lemmas coming from previous models were useful although some properties were still missing, i.e., overridden functions domain and range properties. Nonetheless, we managed to narrow down few of these properties and reduce the number of unsatisfied verification conditions by 10. The fundamental idea behind this proving style is to virtually break down a statement into pieces and consider what basic properties that could be missing.

For instance, the following trivial condition has discharged a large of seemingly unrelated proof obligations in several models.

```
lemma lemma_natural_increment:
forall x, n : int.
mem x bnatural1 /\ n >= 0 ->
mem (x + n) bnatural1
```

A fairly common tactic in the schematic lemma approach, when not familiar with the model, and the condition appears to be true, is to try and identify the few key hypothesis and come up with a lemma that would bridge them to the goal. Although it sounds fairly trivial, proof obligations may contain tens if not hundreds hypotheses so just visually spotting the right few one might be tricky. We are working on heuristics to automatically filter and rank schematic lemma hypotheses.

As an illustration of the finiteness properties consider the following simple example.

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

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

```
lemma lemma_finite_partial_domain:
  forall f : rel 'a 'b, s : set 'a, t : set 'b.
    finite (dom f) /\ mem f (s +-> t) ->
      finite f
```

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

Throughout experiments with a collection of Why3 back-end provers we noticed that not only different provers are better for certain problems, but they also prefer specific style of a writing a lemma. For instance, the order clauses in the conjunction in the left-hand side of an implication may have discernible effect not only on a proof time but also on proof success for some provers. Therefore, it is, to some extent, an experimental process requiring trying out different forms of the same argument.

5 Discussion

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

The models we have chosen for the case study are not particularly large. We have on purpose avoided taking some of the large industry-constructed as they have unusually high proportion of interactive proofs and may argue that Event-B abstraction mechanisms were not used to full extend to manage complexity and reduce the proof workload. In the longer term, however, we would want to tailor our technique to the needs of an industrial user. We believe, and this is supported by our experiments, that with a carefully lemma library and a domain-specific modelling guidance document, industrial user will be able to construct large and useful models without doing a single interactive proof. Failed proof obligations will still be reported, in slightly different style from now, to inform a modeller what is wrong and how it can be fixed. Any proof obligation remaining undischarged after throwing at it all possible automatic provers will be treated as a modelling error irrespective of whether the condition can be potentially proven or not. A similar mindset of restricting the usage of modelling notation in order to gain productivity has been with some great success for the Classical B refinement process [13].

The schematic lemma technique has the potential to significantly alter the way models are proved while proof persistence encourages frequent and deep model refactoring. We also hypothesise that at a certain stage accumulated schematic lemma make automatic proof support so complete that interactive proofs are no longer necessary and an undischarged proof is treated as failed and must be dealt with at a model level.

The idea of generalisation for the purpose of proof reuse has been explored in different settings. Perhaps the most well-known example to aspire to is the tactic or meta-proof language supported by general purpose interactive theorem provers such as Isabelle [21]. It is far more flexible and powerful technique but also requires a different level of expertise from a user. A much simpler technique is having a customisable set of rewrite or simplification rules. In principle, this is offered to some extent by the Theory plug-in; the Atelier-B interactive prover allows a modeller to define custom rewrite rules although this is can be extremely unsafe [6]. Reusable theory components with embedded lemmas, tautologies and rewrite rules are widely used in many verification tools from Maude to ACL2 and also recently available, thanks to the Theory plug-in, in Event-B. Schematic lemmas are far less topical than such theory components but then their inclusion is triggered automatically via syntactic matching rather than through direct instructions from a user.

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

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

Acknowledgments

This work is supported by the RSSB/UK project SafeCap+: SafeCap for integrated optimum capacity, safety and energy strategies at multiple nodes and by the EPSRC/UK project TrAmS-2: trustworthy ambient systems, resource-constrained ambience.

References

1. A. Furst. Event-B model of the Order/Supply Chain A2A Communication. Available at http://deploy-eprints.ecs.soton.ac.uk/129/.
2. J. Bryans A. Iliasov. *Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, chapter A Proof-Based Method for Modelling Timed Systems, pages 161–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
3. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
4. J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
5. A. Fietzke M. Suda P. Wischnewski C. Weidenbach, D. Dimova. SPASS version 3.5. In *Automated Deduction - CADE-22 : 22nd International Conference on Automated Deduction*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.
6. Clearsy. *AtelierB: User and Reference Manuals*. Available at http://www.atelierb.societe.com/index_uk.html.
7. Y. Guyot L. Voisin D. Deharbe, P. Fontaine. Integrating {SMT} solvers in rodin. *Science of Computer Programming*, 94, Part 2:130 – 143, 2014.
8. DEPLOY. Event B and the Rodin Platform. http://www.event-b.org/.
9. C. Marché A. Paskevich F. Bobot, J.-C. Filliâtre. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, August 2011.

10. S. Hallersted. *Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, chapter On the Purpose of Event-B Proof Obligations.
11. T. S. Hoang. Proof Hints for Event-B. *CoRR*, abs/1211.1172, 2012.
12. Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, online at http://www.deploy-project.eu/.
13. L. Burdy. Automatic Refinement. In Proceedings of BUGM at FM'99. 1999.
14. N. Bjørner L. De Moura. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08. Springer, 2008.
15. I. Whiteside L. Freitas. Proof Patterns for Formal Methods. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 279–295, 2014.
16. A. Voronkov L. Kovács. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, 2013.
17. I. Maamria M, Butler. chapter Practical Theory Extension in Event-B.
18. M. Y. Said, M. Butler, C. Snook. Language and Tool Support for Class and State Machine Refinement in UML-B. Available at http://deploy-eprints.ecs.soton.ac.uk/95/.
19. Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.
20. J. Kanig S. Lescuyer S. Conchon, É. Contejean. CC(X): Semantical combination of congruence closure with solvable theories. In *Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198(2) of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.
21. L. C. Paulson T. Nipkow, M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* 2002.
22. T. S. Hoang. Event-B model of the Buyer / Seller B2B Communication. Available at http://deploy-eprints.ecs.soton.ac.uk/128/.
23. The RODIN platform. Online at http://rodin-b-sharp.sourceforge.net/.
24. TPTP. Thousands of Problems for Theorem Provers. Available at www.tptp.org/.