An Outline of a Proposed System that Learns from Experts How to Discharge Proof Obligations Automatically

An outline of a proposed system that learns from experts how to discharge proof obligations automatically *(extended abstract)*

Alan Bundy[1], Gudmund Grov[1,2], and Cliff B. Jones[2]

[1] School of Informatics, University of Edinburgh
{bundy,ggrov}@staffmail.ed.ac.uk
[2] School of Computing Science, University of Newcastle
cliff.jones@ncl.ac.uk

1 Introduction

Many formal methods are "posit and prove" where a designer posits a specification, and then seeks to justify it. This justification is in the form of proof obligations (POs), putative lemmas that need proof. A large proportion of these can be discharged by automatic theorem provers, but there are still some that require user interaction (typically of the order of 5-20%). Discharging these POs can become a bottleneck in the use of formal methods in practical applications, and there are two approaches to dealing with them:

- Follow a *modelling strategy*: change the model/abstraction to a strategy that simplifies the proofs, thus increasing the number of automatically discharged POs.
- Follow a *proof strategy*: accept the challenging POs, and define a strategy for discharging them. Such a strategy must be sufficiently abstract to be able to discharge "similar POs". This is the approach we will take in our AI4FM project. Our aim here is to increase the repertoire of techniques for the proof-strategy approach by learning from proof attempts.

In many cases where a correct PO has not been discharged, an expert can easily see how to complete a proof. We believe that it would be acceptable to rely on such expert intervention to do one proof if this would enable a system to discharge others "of the same form". Specifically, we hope to build a system that will learn enough from one proof attempt to improve the chances of proving "similar" results automatically. By "proof attempt" we include things like the order of the steps explored by the user (not just the chain of steps in the final proof). Thus it is central to our goal that we find *high-level* strategies capable of cutting down the search space in proofs. By separating information about data structures and approaches to different patterns of POs, a taxonomy begins to evolve. A proof (attempt) might be seen to use "generalise induction hypothesis" (e.g. adding an argument to accumulate values) in a specific proof about, say, sequences; a future use of the same PO might involve a more complicated tree data structure — but if it has an extended induction rule, the same strategy

might work. So our hypothesis is: *we believe that it is possible (to devise a high-level strategy language for proofs and) to extract strategies from successful proofs that will facilitate automatic proofs of related POs.*

We have previously outlined the strategy language in [1]. In this paper we attempt to show how the shape of proofs of one result helps in another proof – and also how a proof can be reused in another example. In AI4FM we are not restricting ourselves to one particular formal method, which is illustrated here by providing both VDM [2] and Event-B [3] examples. In §2 we will indicate what we would like to achive; this is followed by an example of tool constraints §3; before we discuss our solution and conclude in §4.

2 Examples indicating scope

This section indicates what we would like to achieve. The examples (taken from the literature) are just sketched here but a technical report version of this short paper will have more detail in appendices.

2.1 A simple example

A trivial teaching example in [2] uses two disjoint sets to record"students who do exercises"; the state in VDM notation is:

$$\begin{aligned} &\mathit{Studx} :: y : \mathit{Id}\text{-}\mathbf{set} \\ &\qquad\quad n : \mathit{Id}\text{-}\mathbf{set} \\ &\mathbf{inv}\ (\mathit{mk}\text{-}\mathit{Studx}(y,n)) \triangleq y \cap n = \{\,\} \end{aligned}$$

A representation of *Studx* could be

$$\mathit{Studx}1 = \mathit{Id} \xrightarrow{m} \{\mathbf{Y}, \mathbf{N}\}$$

VDM reification proofs require that one defines a retrieve function from the concrete to the abstract state. In this case:

$$\begin{aligned} &\mathit{retr}\text{-}\mathit{Studx} : \mathit{Studx}1 \to \mathit{Studx} \\ &\mathit{retr}\text{-}\mathit{Studx}(m) \triangleq \mathit{mk}\text{-}\mathit{Studx}(\{n \mid n \in \mathbf{dom}\ m \wedge m(n) = \mathbf{Y}\}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \{n \mid n \in \mathbf{dom}\ m \wedge m(n) = \mathbf{N}\}) \end{aligned}$$

In order to prove the refinement correct, there are two POs on the types and retrieve function. The first is that the retrieve function must be total — this is trivial. Secondly, the "adequacy" PO has the form:

$$\forall a \in A \cdot \exists r \in R \cdot \mathit{retr}(r) = a$$

This is not quite as simple because it depends on the invariant on *Studx*.

The proof obligations for each operation are (i) the domain rule:

$$\forall r \in R \cdot \mathit{pre}\text{-}A(\mathit{retr}(r)) \ \Rightarrow\ \mathit{pre}\text{-}R(r)$$

and (ii) the result rule:

$$\begin{aligned} &\forall \overleftarrow{r}, r \in R \cdot \\ &\qquad \mathit{pre}\text{-}A(\mathit{retr}(\overleftarrow{r})) \wedge \mathit{post}\text{-}R(\overleftarrow{r}, r) \ \Rightarrow\ \mathit{post}\text{-}A(\mathit{retr}(\overleftarrow{r}), \mathit{retr}(r)) \end{aligned}$$

For such a simple example, we would expect all of these to be discharged automatically or with minimal hand intervention.

2.2 A partitioning reification

The example here is less trivial than that in Section 2.1; the interest is that the first reification step is remarkably similar to the earlier one and thus makes it possible to indicate what we hope to achieve in AI4FM.

This example is derived from Chapter 11 of [2]. The idea is that there is some equivalence relation over X and elements are partitioned according to this relation. Here, the underlying state is a set of disjoint sets. Thus, the state is a partition and is represented by the following type:

$$Part = (X\text{-}\mathbf{set})\text{-}\mathbf{set}$$
$$\mathbf{inv}\ (p) \triangleq \{\,\} \notin p \wedge \forall s, t \in p \cdot s = t \vee s \cap t = \{\,\}$$

If we choose to use a representation of:

$$Keyed = X \xrightarrow{m} Key$$

with the following retrieve function:

$$retr\text{-}Part : Keyed \to Part$$
$$retr\text{-}Part(m) \triangleq \{\{d \mid d \in \mathbf{dom}\, m \wedge m(d) = k\} \mid k \in \mathbf{rng}\, m\}$$

The first steps (adequacy etc.) of the reification proof are (not quite obvious) generalisations of what is done in Section 2.1. Our "ambition" is that the proofs from that section would provide a sufficient strategy to generalise to this case. So here we have an example of the reuse "pattern" of reification proofs being what we hope to achieve.

The actual Fisher/Galler representation can be thought of as a *Forest*:

$$Forest = X \xrightarrow{m} X$$
$$\mathbf{inv}\ (m) \triangleq \forall s \subseteq \mathbf{dom}\, m \cdot s \neq \{\,\} \Rightarrow \exists e \in s \cdot m(e) \notin s \vee m(e) = e$$

This representation follows the Fisher/Galler inspiration that elements are equal iff they have the same root.[3] In this representation a root is therefore a mapping to itself:

$$roots : (X \xrightarrow{m} X) \to X\text{-}\mathbf{set}$$
$$roots(m) \triangleq \{x \mid m(x) = x\}$$

The retrieve function is:

$$retr\text{-}Keyed : Forest \to Keyed$$
$$retr\text{-}Keyed(f) \triangleq \{x \mapsto root(x, f) \mid x \in \mathbf{dom}\, f\}$$

where

$$root : X \times Forest \to X$$
$$root(e, f) \triangleq \mathsf{if}\ e \in roots(f)\ \mathsf{then}\ e\ \mathsf{else}\ root(f(e), f)$$

[3] This representation deviates from [2] with respect to the representation of roots: the type in [2] is "total" ($\bigcup p = X$) and the roots are simply those elements $x \notin \mathbf{dom}\, p$ where $p \in Forest$.

These POs are more difficult than the ones on the first reification because the inductive structure of *Forest* is not obvious. The "ambition" here is that the proofs of the first operation PO would provide a model for those that follow (including those on the operations). This is an example where the way an expert approaches the first operation should hopefully carry over to proofs about further operations.

Another exercise would be to look at the direct development from *Part* to *Forest*.

3 Example indicating tool constraints

This section illustrates current tool constraints. The example is developed within Event-B [3] and the Rodin toolset[4]. It models door lock states (e.g. locked, locking, unlocking, unlocking) – and, similar to §2.1, disjoint sub-sets of door identifiers (type *DOORS*) are represented as a function *doors*, from the type of door identifier *DOORS* into a type *DOORSTATE*, which enumerates all possible states.

In Event-B, the correspondance between the description and actual representation is formalised by a *gluing invariant* (comparable to the retrieve function in VDM), and in this example the gluing invariant is a conjunction – where each conjunct has the "form" illustrated by "the door-locking state":

$$doors\_locking = doors^{-1}[\{DoorLocking\}].$$

Note that *doors_locking* is a (abstract) set of door identifiers. We will focus on the event where a door enters this state, and the description/abstract (left) and representation/concrete (right) events (with all non-relevant parts stripped) becomes:

EVENT lock_door $\triangleq$
  ANY $d, \cdots$
  WHERE
    $d \in doors\_locking, \cdots$
  THEN
    $doors\_locked := doors\_locked \cup \{d\}$
    $doors\_locking := doors\_locking \setminus \{d\}$
    $\cdots$

EVENT lock_door $\triangleq$
  REFINES lock_door
  ANY $d, \cdots$
  WHERE
    $doors(d) = DoorLocking, \cdots$
  THEN
    $doors(d) := DoorLocked$
    $\cdots$

The gluing invariant must be preserved by all events, and the proof obligation expressing this for the invariant over *lock_door* is not discharged automatically by the Rodin automatic provers:

$$\Delta, doors\_locking = doors^{-1}[\{DoorLocking\}] \vdash$$
$$doors\_locking \setminus \{d\} = (doors \lhd \{d \mapsto DoorLocked\})^{-1}[\{DoorLocking\}]$$

[4] see `http://www.event-b.org`.

The proof proceeds by first proving the following intermediate lemma (i.e. applying the cut rule):

$$(doors \lhd \{d \mapsto DoorLocked\})^{-1}[\{DoorLocking\}] = doors^{-1}[\{DoorLocking\}] \setminus \{d\}$$

Both the intermediate lemma and the main goal are then proved automatically by the Rodin predicate prover (PP).

Note that the intermediate lemma could have been avoided by substituting the *doors_locking* term with $doors^{-1}[\{DoorLocking\}]$ using the equality in the hypothesis (this technique is known as *weak fertilization* in rippling [4]). The new goal is then equal to the intermediate lemma (modulo reflexitivity). However, the PP tactic was not able to prove this.

4 Discussion

The major challenge in being able to reuse proof strategies, as illustrated in §2 and §3 (the strategy described in §3 was used on all "similar" gluing invariant POs), is to design a sufficiently general-purpose and robust strategy language so that it can deal with unanticipated proof plans and patches that experts will devise. If we knew in advance what these plans and patches would be, we could include them in the theorem prover, so that the problematic POs would be discharged and would not require expert attention. For example, the strategy used in §2.1 is reused in §2.2.

The strategy language will combine a high-level proof strategy with a "vocabulary" of terms that might be instantiated in the separate theories of data structures stored in the system. The meta-language employed in our rippling/induction proof-planning work [4] provides an existence proof for such a strategy language. We refer to [1] for more details of how we envisage this strategy language.

Acknowledgements This work has been supported by UK EPSRC grants EP/E005713/1 and EP/E035329/1.

References

1. Bundy, A., Grov, G., Jones, C.B.: Learning from experts to aid the automation of proof search. In O'Reilly, L., Roggenbach, M., eds.: AVoCS'09 – PreProceedings of the Ninth International Workshop on Automated Verification of Critical Systems. Technical Report of Computer Science CSR-2-2009, Swansea University, Wales, UK (2009) 229–232 To appear in the EASST electronic publications.
2. Jones, C.B.: Systematic Software Development using VDM. Second edn. Prentice Hall International (1990)
3. Abrial, J.R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press (2009) To be published.
4. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-level Guidance for Mathematical Reasoning. Volume 56 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2005)