Can a system learn from interactive proofs?

Leo Freitas, Cliff B. Jones and Andrius Velykis

Newcastle University
{leo.freitas,cliff.jones,andrius.velykis}@newcastle.ac.uk

Abstract

This paper sets out the on-going research in a project which is investigating how to learn from one interactive proof so that other similar proofs can be completed automatically.

1 Introduction

Formal methods such as VDM [Jon90] or Event-B [Abr10] provide notations to describe specifications and designs. In addition, they offer a way of generating "proof obligations" (POs) that, if discharged, establish that a more detailed design satisfies the properties of a more abstract specification. Our interest is in helping users discharge such POs for developments on an industrial scale. Section 2 provides evidence that, without further help, the use of formal methods by engineers is hampered by their facility with proof tools.

We have embarked on a project that aims to use artificial intelligence (AI) techniques to assist proofs in formal methods (FM). The distinctive feature of AI4FM is that the system we hope to build will learn from a proof constructed by an expert[1] and will be able to apply what has been learnt to other examples. It is of course the case that the task of theorem proving was one of the earliest addressed by AI researchers and their ideas have increased the power of theorem proving systems enormously. In contrast to the discovery of general heuristics, however, the aim in AI4FM is to accept that some proofs will elude any fixed set of heuristics because of the way that complex data structures and user-defined functions interact. We aim to deduce the "why" of an expert's proof and believe that this will be more robust than normal tactics in the sense that it will more readily adapt to other similar proofs and/or survive minor changes in specifications or designs. (It is worth remembering that the second author of the current paper was involved in the project that delivered the mural system [JJLM91] which was an attempt to design a user interface that made it easier to communicate the intent of a proof.)

The main hypothesis of our research is:

> *Enough information can be automatically extracted from a hand proof that examples of the same class can be proved automatically.*

Previous position papers on AI4FM include [BGJ09, GBJI10, JGB10, FJ10]; this paper explores the idea of pre-generating lemmas and exposes our thoughts on an architecture and requirements for the system that we hope to build (as well as setting out the general context).

2 Context

Industrial use of formal methods usually leads to large, yet somewhat structured, models [FW08, CB08, BFW09, DEP12b, DEP12a, DEP12c]. POs about the model consistency and properties

[1] The term "expert" is used throughout this paper as a shorthand for someone trained in proof; there is no intention to underrate the importance of the knowledge of an "engineer" whose expertise lies in the domain of the software being designed.

tend to be repetitive and tedious, and greater automation is key for its successful uptake by industry. This does not preclude the need for an expert to discharge key proofs and indeed to create useful lemmas necessary in the proof process. Within AI4FM, our aim is to learn how to extrapolate from this proof process of an expert working on a few POs, to a set of proof strategies and lemma suggestions for the remaining POs. Due to the structure of models, these POs are often repetitive with potentially only slight differences from previous ones. This gives us hope that we will be able to extract proof strategies and suggest lemmas from our exploration of the proof process as a whole, as well as adapt proofs from previous successful attempts to new lemmas that are structurally similar (*i.e.* lemmas that are syntactically/semantically different, yet share enough meta-level information). We illustrate these strategies below. To set the context of the challenge, we briefly explain a few industrial models with which we have worked.

Within the EU-funded DEPLOY project, there are a number of industrial organisations using the "Rodin tools"[2] to develop small to medium scale industrial applications. Typical figures in one recent application are –of 4215 generated POs– 425 were not discharged automatically; of these, about 400 were deemed to be "very easy" in that they required minimal thought/effort to discharge interactively; a further dozen required less than an hour each; the remaining dozen were troublesome and required more time to discharge.

A more detailed analysis was made of an application from one of the other industrial partners in the DEPLOY project. Here there were about 500 POs generated of which about 80% were discharged automatically. This is useful, but 100 interactive proofs is not an inviting prospect for an engineer. It transpired however that only about five ideas were required to complete all proof tasks. This is precisely the point of the AI4FM project. If it is necessary to enlist an expert, or for one engineer to work for a long time, it is far better if associated proofs do not have to repeat the hand use of the new ideas on similar proofs. We hope that our system will learn the ideas that the expert sees as extra to the fixed collection of heuristics.

For the problem of residual interactive proofs, engineers at SAP (AG) [DEP12c] discovered that such proofs are difficult in that they needed many intermediate lemmas that were closely related to the specific structure and characteristics of the model that they were proving. Such lemmas are not obvious at all but once they have been generated, the difficult proof often becomes much easier. Since these lemmas are closely related to the model under proof and often describe information like control flow or data dependencies, SAP hopes that it can automatically discover such lemmas using static analysis. They have made some progress in this front: for instance, they automatically discovered dependencies among message and control flows for message-choreography models and used them as lemmas to prove consistency and the absence of inconsumable messages. In many cases, all the proof obligations were automatically discharged following the introduction of those lemmas.

Because the first author of the current paper was involved, an even more detailed analysis can be offered of another industrial example: the Mondex Smartcard was commissioned by a bank consortium led by UK Natwest in 1996 [Woo08, p.5-19][3]. The interest was in an autonomous transaction system for electronic cash, where the highest-level of assurance (ITSECL7) was required. It uses formal modelling and proof and led to a hand-crafted formal specification of considerable size [SCW00]: 231 pages of mathematics with 115 paragraphs, and 24 proof obligations. The structure of Mondex entailed rather repetitive proof patterns, as demonstrated by an experiment in a variety of formalisms and provers [Woo08]. The major result was that, regardless of method, (almost) all residual errors were found in each formalism, given a base-line

[2] See `www.rodintools.org`

[3] This whole edition of *Formal Aspects* is given over to a collection of papers using Mondex as a case study for different approaches.

formalisation of the original sources themselves [Woo08, p. 117-139][4].

A key part in the proof process is what we call proof engineering, and although this is proof-system and formalism dependant, it is somewhat transferable given some adjustments. This process involves setting up key (mostly trivial) lemmas linking various aspects of the specification. For instance, notions about types and how they are related with respect to the main operators used (*e.g.* a sequence of numbers is a function from numbers to numbers); explicit information about inductive datatypes not automatically given by the proof system (*e.g.* injectivity and disjointness properties in the appropriate shape); automatic rewrite rules setup (*e.g.* what lemmas are to be automatically used by the term simplifier); equational reasoning (*e.g.* theorem provers usually choose equations from left to right); refactoring and reshaping of definitions and types without loss of meaning (*e.g.* tools have preferred *modus operandi*, which makes little sense to fight against — instead one needs to provide auxiliary definitions, lemmas and proofs to show the rearrangement has preserved meaning); *etc.* Although mostly trivial, these proofs are crucial in hinting to the proof system how more involved proofs are to be handled. Luckily, the process is fairly deterministic, given a certain proof system and the desired shape for the model involved.

The proof engineering needed for the base-line mechanisation increased the number of paragraphs to 199 and proof obligations to 318. These 84 extra definitions were used to gear the proof process according to what was required by the theorem prover at hand, and this is a recurring necessity in our experience. Fortunately, these extra definitions were (mostly always) trivial refactorings, with a couple of new difficult proofs. These new proofs turned out to be about something missing in the original (hand-crafted) model regarding presumed finiteness properties. Furthermore, the scrutiny of mechanisation uncovered that a whole family of proofs were elided (see [Woo08, p. 129]), hence the big increase in the number of proof obligations from 24 to 318. Of these, 64 were due to type-morphism needs during proof engineering, and had simple, if not trivial proofs. An extra 72 lemmas were needed in order to decompose the problem into amenable chunks for the prover to handle, and to refactor parts of the effort to enable proof reuse. An extra five general lemmas were needed for the mathematical symbols present. These have involved proofs, yet they are normally reusable on different problems using the same symbols. The remaining proof obligations were divided into specification consistency (*i.e.* 66 proofs about functions applied within their domains), and model feasibility (*i.e.* 81 precondition proofs/lemmas for operations). Six extra proofs were needed to show the consistency of the few axioms involved. The overall formalisation had 199 paragraphs, 318 proof obligations, and 4544 proof steps. The proof effort took about a month[5] and was divided into three stages: specification (16.7%), feasibility (66%), and refinement (17.3%). During specification, proofs are easier and relate to internal consistency of the model itself. The feasibility proofs were the largest, yet not the most complicated part, where the refinement proofs were complex. They were large mostly because of the size of the model[6], yet they were repetitive with only minor differences. This is the good news: we claim repetitive proofs are amenable to further automation through learning the proof strategies involved — this is our goal in AI$_4$FM.

One must of course pause and ask whether some particular theorem proving system is not smart enough to blow away all such POs. Quite apart from the relevant undecidability properties, the sheer scale of something like the specification of the iFACTS air traffic control system (rumoured to have a four digit number of pages of Z) ought to make one doubt that any completely automatic system will ever suffice. We have couched the main hypothesis in

[4] All proof source material is available at `vsr.sourceforge.net`

[5] Having the hand proofs from [SCW00].

[6] When fully expanded, one proof goal led to about 45 pages of A4!

terms of interactive proof but recognise both that such human guided proofs often invoke "sledgehammer" style aids and that even fully automatic proofs can contain useful pointers as to what techniques work on a class of proofs. Such information will certainly not be ignored but our initial attack is to focus on learning from the interaction the user creates.

3 What does an expert do?

There is of course no simple answer to a question as general as "What does an expert do?" even when the context here limits the application of expertise to discharging recalcitrant proof obligations. The following paragraphs identify some functions and how we hope the system we intend to build can learn from the particular displays of expertise.

The scenario is one where hundreds (if not thousands) of POs are generated and the user needs to come up with ways to discharge them. In practice, a number of these POs can be discharged automatically by the prover itself or with some naive strategy. The remaining POs require some effort in tuning the prover as well classifying their nature (*e.g.* is it a well-formedness, feasibility, or refinement PO?). Next, given this classification of POs, certain known strategies are possible. For instance, feasibility POs often involve finding a witness satisfying an invariant, which does require some effort. Finally, the residual (more complex) POs are the ones where the expert is really needed. We think our strategy elicitation process will take place at several of these stages. And with enough proof engineering in place, we usually have earlier success, hence the expert can take more time to handle the really important/difficult proofs.

Naive strategy. In a naive (step-by-step) strategy, specific definitions are unfolded and the "heaviest" proof tool available is thrown at the problem. This is useful and often succeeds. That is providing the goal to be proved already has at its disposal the proof engineering features needed. It is important to highlight contextual assumptions regarding available proof engineering, because it can be the source of great frustration during (failed) proofs: "why on earth hasn't the goal proved *true* now, when in a slightly different case it did?" For instance, when proving goals involving records or tuples (of varied size), this often happens: a goal over the explicit tuple (x, y) proves true, whereas one with a tuple-typed variable p does not, despite the fact $p \simeq (x, y)$. This is an instance of a missing lemma informing the prover that the models talk about tuples in two different formats interchangeably. The key point here is that failure **ought** to provide better feedback to the user on the reasons why it came about. We elaborate on this in the next section.

Generic strategy. One of the most important aspects of industrial proofs is the need for so-called proof-engineering: the adjustment/refactoring of given models in order to enable them to be processed by the theorem prover. Although such a process is theorem prover specific, the generic strategy is not, and can be divided into categories like: type-morphisms linking known theories with current representations (*e.g.* some record with relations that are chained together in the invariant to be linked to a theory of transitively closed relations); rule inference setup (*e.g.* determining what is to be picked as introduction, elimination or simplification rules will influence the levels of automation); early case splitting and/or induction over appropriate variables; *etc.*

Tuning models for proof. An expert will have developed a "taste" for structuring the stepwise development from a formal specification, but this is not to say that all experts would

agree. It is interesting how different styles are even before formal proofs are considered. For example, [HJN94] points out that there is a "Z-style" [Hay93] in which redundancy between fields of, for example, states is embraced whereas it is more common in VDM [Jon90, JS90] to reduce the need for data type invariants by minimising redundancy and define auxiliary functions that can extract derived information. Once formal –machine governed– proofs come into the picture, the range of differences increases. Most experts have a notion of introducing "one design decision" per step of development — in other words, it is not thought wise to make a single step from a (large) specification to detailed code and generate POs that attempt to bridge the gap. The main argument against such a move is that any errors are difficult to spot — attempting large proofs is not a good debugging strategy. But there is another extreme position and that is where an expert faced with a collection of POs that are not discharged automatically is prepared to introduce another step of development to simplify the proof task. This is not the place to take a position on whether this is, or is not, a good move but just to record that it is an option. Research aimed at automating such detection is being undertaken in [IGLB11].

Formulating lemmas. Most experts will attest to the importance of lemmas in tackling large proofs; perhaps the earliest call for their importance in increasing the productivity of program verification is [Dah78];[7] [Jon79] picks up the theme as "developing a theory of the data types". The ACL-2 school [KMM09] is explicit about the use of lemmas to guide automatic proofs. It is clear that experts are good at looking at a problem and formulating useful lemmas. In the best cases, this process is actually productively done even before POs themselves are tackled. We can add to our overarching research hypothesis:

> **mini-hypothesis:** *Given the type and function definitions* and *proof obligations, useful lemmas can be generated that will simplify (possibly to fully automatic) the discharge of the POs (we believe that this applies to realistic –industrially interesting– examples)*

For instance, from a successful proof of a lemma involving the domain of a relation ($x \in \mathbf{dom}\, R$) and the need for a similar lemma where we know only about $(x, y) \in R$, the adaptation is to suggest the range of the relation ($y \in \mathbf{rng}\, R$). This topic is central enough to our approach that the next section is devoted to an illustrative –albeit small– exploration.

4 A small, but representative, problem

It has been made clear above that the target of the AI4FM project is to contribute to the productivity of engineers working with large industrial specifications and developments. It is however possible to learn lessons from (larger) textbook examples and this section draws some conclusions about the extraction of lemmas from just such an example — the lessons are focused on the determination of lemmas — the example is the recording of equivalence relations and its implementation using the Fisher/Galler tree data structure.

Before coming to the equivalence relation problem, it is worth looking at an even simpler problem. It has been observed that spotting lemmas over recursive data structures can overcome the way in which theorem provers get blocked in inductive proofs. In the case that types are unrestricted by predicates, the "word algebra" of the generators gives a clear indication of how inductive proofs might be conducted. Trivial examples are straightforward but, even with a

[7] Cooper's early [Coo66] is a little appreciated paper.

type as simple as sequences, inductive proofs can get stuck. An oft-quoted example is proving that the reverse of the reverse of a sequence is the original sequence. Given:

$$
\begin{array}{l}
rev : X^* \to X^* \\
rev(s) \quad \triangleq \\
\quad \textbf{cases } s \textbf{ of} \\
\qquad [\,] \to s \\
\quad [hd] \frown tl \to rev(tl) \frown [hd] \\
\quad \textbf{end}
\end{array}
$$

the task[8] is to show $rev(rev(s)) = s$. To an expert, it is clear that it is useful to have a lemma to make the induction work:

$$rev(s1 \frown s2) = rev(s2) \frown rev(s1)$$

The need for a lemma can, for example, be spotted by a *proof failure analysis* [IGLB11] and *rippling* [BBHI05] can even spot the required lemma.

In fact, the situation above is both easy to generalise and possible to detect *before* there is a failing proof to be analysed. It is a consequence of the form of the recursive part of the definition of *rev* that any term of the form $f(rev(s))$ is going to give rise to the need for knowledge about $f(\cdots \frown \cdots)$. Considering something like $\textbf{len}\, rev(s)$, it is not difficult for an extension of IsaCoSy [JDB11] to generate $\textbf{len}\,(s1 \frown s2) = (\textbf{len}\, s1 + \textbf{len}\, s2)$ from among the operators that could combine two integers, given:

$$
\begin{array}{l}
len : X^* \to \mathbb{N} \\
len(s) \quad \triangleq \\
\quad \textbf{cases } s \textbf{ of} \\
\qquad [\,] \to 0 \\
\quad [hd] \frown tl \to 1 + \textbf{len}\, tl \\
\quad \textbf{end}
\end{array}
$$

Yet, how would this consideration surface? How would a system determine that $\textbf{len}\, s$ is the/a useful definition to pick? In addition, there is the question of how to actually define the lemma above that we hope will be useful? We answer these questions below.

First, we would like to introduce a more interesting problem of the "equivalence relation". The specification in [Jon90, §4 and §11] of a system that records equivalence relations is built around the intuitive data structure that stores all equivalent elements (from some underlying, but finite, set X) into sets within an overall set of sets:

$$Partition = \{p \in (\mathbb{N}\text{-}\textbf{set})\text{-}\textbf{set} \mid inv\text{-}Partition(p)\}$$

Where:

$$
\begin{array}{l}
inv\text{-}Partition : (\mathbb{N}\text{-}\textbf{set})\text{-}\textbf{set} \to \mathbb{B} \\
inv\text{-}Partition(\,p) \quad \triangleq \quad is\text{-}prdisj(\,p) \wedge \{\,\} \notin p \\
\\
is\text{-}prdisj : (\mathbb{N}\text{-}\textbf{set})\text{-}\textbf{set} \to \mathbb{B} \\
is\text{-}prdisj(ss) \quad \triangleq \quad \forall s_1, s_2 \in ss \cdot s_1 = s_2 \vee is\text{-}disj(s_1, s_2)
\end{array}
$$

[8] *rev* is defined by case-analysis on a sequence X^* using concatenation ($\_ \frown \_$).

It is immediately obvious that the elements of this type are not freely generated in the same way that X^* forms a word algebra; the predicate restriction is crucial. Turning to the operations of the specification, with this model, it should be straightforward to show that the initial configuration

$$p_0 = \{\{x\} \mid x \in X\}$$

is within the type ($p_0 \in \mathit{Partition}$) and that an operation for merging the equivalence classes of two elements (say, a, b)

$$p' = \{s \in p \mid a \notin p \wedge b \notin p\} \cup \{\bigcup \{s \in p \mid a \in p \vee b \in p\}\}$$

preserves the type *Partition*.

The difficulty with performing mechanical proofs of this latter property is that the definition of *Partition* given above does not provide a natural induction. In this case, it is not difficult to define a way of generating all elements of *Partition* but it is non-trivial to prove that it coincides exactly with the given definition. (More precisely, it is straightforward to show that all inductively created elements satisfy the property above but harder to show that all elements of the *Partition* definition are created by the generators.) This matters because some of the POs for data reification in VDM [Jon90] quantify over the abstract type.

The situation becomes more difficult when the clever Fisher/Galler tree representation of equivalence relations is addressed. In [Jon90, §11],

$$\mathit{Forest} = X \xrightarrow{m} X$$
$$\mathbf{inv}\ (m) \triangleq \forall s \subseteq \mathbf{dom}\, m \cdot s \neq \{\} \ \Rightarrow\ \neg(\mathbf{rng}\,(s \lhd m) \subseteq s)$$

It is then possible to define:

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

That this function is total over *Forest* (but not over arbitrary $X \xrightarrow{m} X$) follows from the invariant.

The empty *Forest* is:

$$f_0 = \{\}$$

and satisfies *inv-Forest* because the only $s \subseteq \mathbf{dom}\,\{\}$ is $\{\}$ which vacuously satisfies the implication.

Unfortunately, our experiments with proofs using either Z/EVES [Saa97] or Isabelle [NPW02] on the definition of *Forest* above show that its invariant is extremely inconvenient.

The first author of the current paper has developed a set of lemmas about relations and transitive closure that were used to characterise an equivalent version of the invariant of forests that was amenable to inductive proofs. The exercise exposed interesting properties about a forest element "depth", as well as issues about the way to calculate the root of an element. These type-morphisms are key in the proof process and are theorem prover dependent. Similarly, an inductively defined version of the problem is given in Isabelle/HOL by our AI4FM colleagues in Edinburgh. This HOL version implies the original definition (*i.e.* refines it), and is amenable to inductive proofs. Showing that they are equivalent requires a rather involved proof, though. Both exercises have their shortcomings, and somewhat departed from the original definition of *Partition*. However, it was possible to prove some (refinement) links with the original problem. This brings an interesting observation that the formalisation geared towards mechanisation might look rather different than one that is "intuitively" presented.

Thus, the message is that the role of lemmas in proof is crucial. They are needed as either: an alternative (morphisms of original) definition that is more amenable to proof; a reformulation of the problem in some cases where errors are uncovered; a stepping stone towards the goal itself (*i.e.* so-called weakening lemma); and so on. Without these, some proofs become unnecessarily long and cumbersome, whereas in most cases, completing the proof is just infeasible. So, detecting the right "shape" of the lemma is one key measure of success. From the simple example above on *rev*, we conjecture that machine learning techniques of concept formation and deduction [JDB11] can be used to both rule out ineffective lemmas, and to suggest useful ones.

These lemmas usually come in two shapes: as (i) paramodulation rewrite rules ($lhs = rhs$)[9], or as (ii) conditional rewrite rules ($lhs \longrightarrow rhs$). How will our tools infer/suggst each of these parts? In (i) we appeal to definitions of terms involved in the current proof goal together with meta-information from definitions in order to generate the *lhs* (*i.e* the part to match the current goal with). The *rhs* is more difficult to infer/generalise. The difficulty here is regarding the potential explosion of possibilities given the (function) symbols involved and their definitions. This is where the role of meta-information is crucial, and that includes both positive and negative information about the meta-level reasoning properties of a definition. For instance, in the example above on **len** $rev(s)$, the "symbols of interest" (†1) are: **len** , rev, $\_ \overset{\frown}{} \_$, $\_ + \_$. Suppose that we annotate the definition of $(\_ + \_)$ as commutative, where the same can be said negatively about $(\_ \overset{\frown}{} \_)$ (*i.e.* $\neg$commutative). These meta-level properties can be given weights or worthiness, and PROLOG-style descriptions akin to what is described in [Bun83, Parts. 3, 4]. Say we also annotate the definition of **len** s as associative in the sense that **len** $(s1 \overset{\frown}{} s2) =$ **len** $(s2 \overset{\frown}{} s1)$. Turning to the goal **len** $rev(s)$, we look into the definition of $rev(s)$ and note both $(\_ \overset{\frown}{} \_)$ and case-splitting on the X^* data type as the "dominating operators" (†2) and that its domain type matches with that of **len** s, whose definition is dominated by the operators $(\_ \overset{\frown}{} \_)$, $(\_ + \_)$ and case splitting over X^*. This suggests the left-hand-side of a (potentially useful) lemma for the goal **len** $rev(s)$ will be **len** $(s1 \overset{\frown}{} s2)$. Next, for the right-hand-side, we "filter out the symbols" (†3) with conflicting meta-information or symbols we are already aware of, which leads to $(\_ + \_)$ that is associative as **len** is with respect to concatenation. We see the role of these (†1-3) terms as some of key parts from which we will be able to learn from an expert the sort of lemma required.

From the library of lemmas involved, other symbols will crop up as associative or additive or multiplicative and extra (spurious) suggestions are likely to be made. Given we are lucky and the result is non-empty, we suggest a equivalence lemma like

$$\text{?Lemma L1:} \quad len(s1 \overset{\frown}{} s2) = len(s1) + len(s2)$$

This conjecture is then passed to an available theorem prover (probably running in the background) to spot any counter-example, which would filter the whole lemma out as spurious. Other spurious conjectures could involve multiplication or exponentiation, say. From the context or indeed given user input/feedback, the resulting lemmas are the "right" ones suggested to be proved. When there are no matches after filtering, we will need some help from the user. Yet, at least for the left-hand-side we have confidence it will be useful. Furthermore, we could apply the same meta-level reasoning to conditional rewrite rules, yet their role in proof is slightly different. In backward proof, the *rhs* matches the goal conclusion that is substituted by the (usually weaker/simpler) *lhs*, whereas in forward proof, the *lhs* matches some hypothesis of the goal that gets substituted by the (usually stronger) *rhs*.

[9] Theorem provers have a preferred *modus operandi*, usually left-to-right when applying lemmas.

Even if the suggested lemma is not spurious and looks promising, it **must** be in a shape that matches how the underlying theorem prover simplification/rewriting engine works. Otherwise, although the lemma is "right/useful", it will not be picked up by the prover automatically and instead would require (often) cumbersome manual instantiations. Thus, it is crucial that "local knowledge" is applied when generating the lemma suggestion. For instance, in the Z/EVES theorem prover, almost all machinery is geared towards tuples (x, y). If definitions involve triples or n-ary tuples (x, y, z), any suggestion is almost certainly ignored by the prover automation processes. Instead, one ought to produce paired tuples like $(x, (y, z))$ or $((x, y), z)$, where the associativity is not relevant. Similarly, for the Isabelle/HOL prover, cross-products are defined as a datatype, and hence are amenable to case-analysis and induction. That also means a variable p that is a tuple will not match directly with explicit tuples like (x, y). So, in a definition like $swap(x, y) = (y, x)$, the lemma that $swap(swap(p)) = p$ is surprisingly not straightforward to prove. That is mostly because p does not match (x, y) from the definition of *swap*. Here is another lesson from both (if not all) provers: in declaring lemmas, always take into account the shape of the definition of function symbols involved in order to make sure it will appropriately match during term rewriting/simplification.

Ideas similar to these were systematically applied by hand in the *Forest* example described above. After ruling out spurious cases, interesting lemmas were found. They were in three categories: proof engineering lemmas relating the new datatype to mathematical structures (*e.g.* sets, relations, closures) known to the prover(s); the ones useful for the goal at hand; and the ones involving new notions not included in the problem itself, like the possible inductive definition of the depth of an element being useful in characterising a *Forest*, as opposed to chasing its root upwards as in the original definition. These were not found directly, and required some interaction. They were useful to establish a equivalence between *Forest* defined in terms of the transitive closure and defined in terms of depth. With that, we could choose the most suitable representation when handling goals involving *Forest*. Details of this exercise can be found in [FJ11].

A crucial problem of this kind of exercise is how and with what to annotate the definitions of interest with meta-level information. Fortunately, formal methods models are often restricted to a reasonably small set of mathematical operators (*e.g.* on sets or relations or maps), together with user-defined record/datatypes. So, we do not see the initial setup of meta-level information as too much of a burden, and we do expect the user to provide (some of) such meta-level data to new definitions/types that are non-trivial to infer. Obviously, the mathematical operators/libraries used also need to be annotated when inference is not possible, but that is done once per prover system.

In practice, we are using an unsophisticated filtering process that tries to categorise lemmas into type-morphisms, rewrite rules, *etc.* The idea is to improve this process and produce an evidence-based approach showing the usefulness of lemmas based on where they are applicable within the proof process as a whole. Thus, there are a few aspects to consider: the proof engineering associated with an expert's "local knowledge" of both the problem and the prover; our tools learning potential (new) "local knowlege" to extract from current proofs (*e.g.* equivalence of (new) record types being just "like" the problems with tuples above); and more sophisticated techniques like *rippling* to tackle the harder residual problems. In the next section we discuss an architecture for our current tools and how it interacts with some theorem provers.

5 A possible architecture

In the main hypothesis we propose that enough useful information can be extracted from an expert's proof to facilitate automatic discharge of similar proof obligations. The whole problem can be viewed as a three step process:

1. *Collect* information about an expert's proof;
2. *Extract* proof strategies;
3. *Replay* proof strategies to discharge POs.

In this section we propose ideas about an architecture of a system to perform such steps. We go over the steps in reverse order, investigating what is needed from earlier steps. In this manner some requirements can be established about types of information which must be captured from an expert's proof.

Replaying proof strategies. If we assume that appropriate strategies can be extracted, then there are a number of techniques to replay them. A prominent one is *proof planning*, a framework for representing and automatically applying common patterns of reasoning (*e.g.* the proof strategies) [Bun88]. *Rippling* [BBHI05] is a powerful proof planning technique, which applies available rewrite rules based on structural features of the hypotheses and the goal. Heneveld [Hen06] suggests encoding *detectors* of certain structural proof features, which would signal when a strategy (schema) should be applied. Different weights could be provided and/or learnt to determine the most applicable strategy if several features match. These techniques are supported to some extent by a number of proof planning tools, such as IsaPlanner [DF03], ΩMEGA [BCF+97], Feasch [Hen06] and others.

Extracting the proof strategies. The power and usefulness of replay techniques depends on available proof strategies. These are customarily developed by hand and then fed to automatic search algorithms to discharge proofs. As set out in the main hypothesis, we hope to extract such proof strategies from the expert's proof. For each strategy, we want to know *why* (*when*) was the strategy chosen, and *what* constitutes the strategy. Strategies should be generalised for a certain class of POs. However, we do not expect to always have generic strategies—we envisage domain-specific proof strategies as well.

Some advances have been made in extracting proof strategies by data mining existing proofs [JKPB03, DBL+04]. This approach follows an assumption that certain sequences of low-level proof steps (*e.g.* rewrite rule applications) form reusable strategies—*i.e.* if a similar problem is encountered, the expert would take the same proof steps. Therefore data-mining techniques are employed to find the common patterns of proof-step sequences. The new tactics have success when being applied to an already known family of proofs, from which the learning group of proofs originated. Furthermore, they can be used to suggest what proof step should be taken next, based on proof steps already written.

On the other hand, data-mining proofs has several shortcomings. Capturing when the extracted tactics need to be applied is difficult. Also, data-mining requires a corpus of available proofs to learn from. From the AI4FM project's perspective, we would not have the luxury of a large number of expert proofs available for every new problem. We expect our system to start assisting the user as soon as the first proof is completed. From this single proof (or a small number of them at most) we expect to extract the proof strategies and start discharging the remaining POs of the same "family". It would also be infeasible to count on theorem prover

core libraries to provide the proofs for data mining, since they are customarily "cleaned-up" and trimmed to the absolute minimum. If one existed, a "central online repository of general proof strategies" could facilitate data mining, but would probably not have much success when applied to specific domains and problems.

Thus we expect to start with a single proof, possibly with multiple (possibly failed) attempts. To extract a proof strategy successfully from a single proof, the proof attempts need to be annotated with "why" information (*e.g.* why certain proof steps were taken, what proof features triggered this line of reasoning, etc.). Such "why" annotations must capture the expert's proof process adequately — we examine this below. From this information we expect to obtain a high-level proof process –backed by specific proof steps– which could be generalised for that family of POs.

During an attempt on a similar PO we would try matching the new proof to the captured proof attempt. We hope that the captured high-level proof strategies with appropriate "why" information about goal terms, conclusions, used lemmas, proof tactics and other proof data will help drive the automatic proof search during the new attempt.

Capturing the proof process. We aim to capture the expert's proof process in order to reuse the experience in attempting to discharge similar POs. When using "waterfall" (chain of tactic applications) style theorem proving, like in Isabelle/HOL[10] [NPW02] or Z/EVES[11] [Saa97], the final proof is usually unstructured and low-level. Furthermore, the developers strive to "clean-up" the proof after their initial success, thus losing many details of how it was reached, as well as how one recovered from failure, which is crucial to us. Isabelle/Isar [Wen02] can be used to structure the proof by defining explicit intermediate hypotheses and goals, however the overall proof is still too low-level. The proof planning system ΩMEGA utilises a richer proof data structure [ABD$^+$06], which accounts for different granularity and alternative proof attempts (*e.g.* a proof can contain different attempts of low-level tactic steps and an alternative high-level proof plan).

The aim of these systems is to produce and verify the final script of a proof obligation. For this reason, they are not concerned with extra details. We expect our system to capture and store the expert's *proof process*, including additional "why" information such as:

- Proof *granularity*: we want a high-level proof plan as well as the tactic steps. The proof should be captured at any arbitrary granularity that the expert deems appropriate.

- Proof *structure*: while the proofs are usually done as a sequence of steps, the actual structure is rarely linear (*e.g.* the base and step cases of inductive proofs can be seen as branches in proof trees).

- Multiple proof *attempts*: we want to capture the whole proof development. Failed attempts may still contain generally applicable proof steps, even though they were not successful in that particular case. All versions leading to a finished proof should be captured.

- *Intent*: high-level explanation of why the expert took specific proof steps. These could be simple tags, giving names to underlying proof reasoning (*e.g.* akin to *Isolation*, *Collection* and *Attraction* in [Bun83, Ch. 12]). The *intent* would be coupled with *properties* (below) to paint the whole proof process picture.

[10] The LCF family of theorem provers.
[11] The Boyer-Moore family of theorem provers.

- Proof *properties*: anything that drives the expert's proof. It can be the shape of the goal, the available lemma, certain datatypes or records, *etc.* Such information is usually readily available during the proof process, however deducing it from the finished proof is difficult. We aim to generalise the proof properties to form parts of the strategies as they would govern when the extracted strategies apply in the next proof attempt.

  The properties are similar to *features* in [Hen06]. The features, however, were mostly left to describe basic term shapes in the hypotheses and the goal. Furthermore, we want to give first-class treatment to lemmas used in the proof steps (lemmas are actually at the core of ACL2-school provers [KMM09]). We also expect most properties in large (industrial) proofs to be related to identifying specific fields in large data structures and reasoning about them. Therefore we aim to have full support for record-like structures.

Starting from different attempts on the small (but sufficiently complex) example in Section 4, we can begin analysing what information should be captured to describe the proof process. We are interested in whether we can generalise the process and what parts of it can we reuse in the next proof. We expect to get different processes for every expert attempting the problem, depending on his proving style and experience.

Proof process of automated tactics. Current theorem provers have powerful automated tactics, which can complete an outstanding proof in one step or advance it by sophisticated transformations. These include advanced simplifiers, reasoners, tableau provers, or combinations of these.

When applied during proofs, these methods have different uses, depending on the approach the expert is taking. For instance, they can be naively (blindly) applied to a complex goal just to check whether it can be solved automatically or advance it to a state where the expert can take over. In other cases, the expert may have followed a high-level (abstract) proof plan and taken the proof to a state where (s)he is confident that the automated tactic can finish it off. The tactic is then used to avoid manually entering all intermediate steps.

Such semi-interactive theorem proving is a common approach to discharging POs. However, the versatility of automated tactics means that reusing them in similar proofs may yield different results (or fail altogether) depending on the available proof context. When capturing the proof process involving such automated tactics, we would like to get insight into why the tactic has been used, what it has done and what is needed to achieve similar results during its reuse.

Sadly, the inner workings of such tactics are usually impenetrable, short of examining their source code. This is quite different from automatic *proof planning* tactics, which are backed by high-level proof plans. The simplest approach is treating the automated tactics as black boxes. Even then, however, we may want to check where they have been used and how they have affected the goal—a proof process may be hidden there, *e.g.* a tactic is only applied in certain situations. Other tactics may output a trace, which carries some information on its application. For example, we could infer what shape/definition lemmas would be useful by analysing what lemmas have been applied by a simplifier tactic. When reusing the proof for a similar PO, we would know that similar (if not the same) lemmas are needed when running the simplifier.

Another crucial problem is then to identify how suggested lemmas are being used by the prover, if they are being used at all. One source of inspiration is to inspect successful lemma use from the prover's proof-trace. Because simplification chains can take a considerable number of automatic lemma applications, we also consider sieving/mining through this data as a machine learning exercise of a different kind [JKPB03, DBL$^+$04]. One strategy to succeed in such an effort is to look inside the most used automatic proof procedures of theorem provers of interest.

In Z/EVES, we have *prove by reduce*, whereas in Isabelle we have *auto*, *sledgehammer* and others. By looking into how these tactics are encoded and by unpicking each step involved, we try to infer what part of the goal subsystem these tactics will influence. With this setup, next we inspect the proof trace at different stages, and by knowing how the provers build their proof contexts, we can make better predictions at the time of collecting information on whether/when a lemma will be useful or not. We hope this will also help us in deciding the shape (*lhs*/*rhs*) of lemmas discussed above. For instance, if we need some lemma about *len*, $(\_ \overset{\curvearrowright}{} \_)$, and $(\_ + \_)$, we could look at previous (successful or failed) proof attempts involving these symbols by such automated procedures to identify the best pattern to use when suggesting new lemmas. Conversely, if the user provides an abstract proof plan and/or "why" annotations at various stages of the proof attempt, we could try to match the expectations from this high-level to the actual lemma being suggested.

Within the AI4FM project we hope to be able eventually to handle different proof process styles by various experts using a variety of theorem proving systems. The success of extracting and reusing the proof strategies would depend on understanding the available tactics, reading information provided by the theorem prover, as well as the expert providing necessary initial hints.

High-level architecture. We are currently building a modular AI4FM system which assists users with their proof but does not force them to work in a prescribed way. We capture the proof process by "wire-tapping" the proof assistant. By "listening" to user actions and responses from the theorem prover we can record basic information, such as results of tactic applications, how does the proof evolve, etc. The activities of our system should be as invisible as possible, but initially we expect the user to tell the system about the proof process, *e.g.* what is being done and why.

The captured information will be stored in a *proof process repository*. Then, all other applications would be built on top of this data store as different components. We envisage various uses for the collected data:

- Extracting and replaying proof strategies.
- Inferring the proof process. The captured data should feed back into the capturing process, which would allow more automation in understanding what the expert is doing.
- Proof analysis. By capturing the proof process and having feedback from replayed strategies, we could amass a proof corpus to perform statistical analysis or other queries.
- Visualisation. Visualising the proof process—which would include the attempt history, various granularity and intent—could assist the user with proof exploration and development.
- Comprehension. The captured proof process would help in understanding other people's proofs. This would be useful for an expert coming back to an old proof, inspecting other people's proofs, or for educational purposes: a student would benefit from "watching" an expert's proof process. The benefit of different levels of granularity in *proof explanation* is recognised in [Fie01, ABD+06]; adding the proof development history and various "why" information takes it even further.

A modular architecture separates the concerns of data capture and its usage. We aim to provide formal models of the proof process and clear dependencies in our architecture, thus inviting contributions and extensions from AI and formal methods researchers.

6 Conclusions

The general aim to increase automation in the discharge of formal methods proof obligations is shared by many projects because the payoff is clear. We support all such endeavours and will gladly adopt their results. Our approach is not however just to make systems that are smarter themselves. We are focusing on how to design a system that can learn from tracking the process used by an expert. The (reasoned) expectation is that proof tasks can be grouped and that an idea that works for one member of a group can be captured and reapplied to other tasks in the group so that previously failing proof tasks get discharged automatically. An outline architecture for the capture phase is given above.

Further avenues of exploration include checking what can be learnt from proof attempts that become completely blocked and helping debug wrong conjectures.

Acknowledgements

Members of the Edinburgh part of the AI4FM project (Alan Bundy, Gudmund Grov and Yuhui Lin) contribute to all of our discussions and this stimulus is acknowledged as are the discussions with Andrew Ireland and Maria Teresa Llano Rodriguez of Heriot-Watt university. We also gratefully acknowledge the EPSRC funding for AI4FM.

References

[ABD+06] Serge Autexier, Christoph Benzmüller, Dominik Dietrich, Andreas Meier, and Claus-Peter Wirth. A generic modular data structure for proof attempts alternating on ideas and granularity. In Michael Kohlhase, editor, *Mathematical Knowledge Management*, volume 3863 of *Lecture Notes in Computer Science*, pages 126–142. Springer Berlin / Heidelberg, 2006.

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

[BBHI05] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[BCF+97] Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Wolf Schaarschmidt, Jörg H. Siekmann, and Volker Sorge. OMEGA: Towards a mathematical assistant. In William McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 252–255. Springer, 1997.

[BFW09] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, 2009.

[BGJ09] Alan Bundy, Gudmund Grov, and Cliff B. Jones. An outline of a proposed system that learns from experts how to discharge proof obligations automatically. In *Proceedings of Dagstuhl Seminar 09381: Refinement Based Methods for the Construction of Dependable Systems*, 2009.

[Bun83] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 4th edition edition, September 1983.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120. Springer Berlin / Heidelberg, 1988.

[CB08] David Copper and Janet Barnes. Tokeneer ID station EAL5 demonstrator: Summary report. Technical Report S.P1229.81.1 Issue: 1.1, Altran-Praxis, August 2008.

[Coo66] D. C. Cooper. The equivalence of certain computations. *BCS, Computer Journal*, 9:45–52, 1966.

[Dah78] O-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *EEC-Crest Course on Programming Foundations*, pages 57–114. IRIA, 1978. Also printed as Technical Report 33 of Institute of Informatics, University of Oslo.

[DBL+04] H. Duncan, A. Bundy, J. Levine, A. Storkey, and M. Pollet. The use of data-mining for the automatic formation of tactics. In *Workshop on Computer-Supported Mathematical Theory Development*. IJCAR-04, 2004.

[DEP12a] DEPLOY. Enhanced deployment in the automotive sector (WP1). EU Project Deliverable D38, DEPLOY Project, 2012.

[DEP12b] DEPLOY. Enhanced deployment in the space sector (WP3). EU Project Deliverable D39, DEPLOY Project, 2012.

[DEP12c] DEPLOY. Report on enhanced deployment in business information sector. EU Project Deliverable D42, DEPLOY Project, 2012.

[DF03] Lucas Dixon and Jaques Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[Fie01] Armin Fiedler. Dialog-driven adaptation of explanations of proofs. In Bernhard Nebel, editor, *IJCAI*, pages 1295–1300. Morgan Kaufmann, 2001.

[FJ10] Leo Freitas and Cliff B. Jones. Learning from an expert's proof: AI4FM. In Tom Ball, Lenore Zuck, and Natarajan Shankar, editors, *UV10 (Usable Verification)*, November 2010.

[FJ11] Leo Freitas and Cliff Jones. Mechanising the Fisher/Galler problem in Z/Eves. Technical report, Newcastle University, 2011.

[FW08] Leo Freitas and Jim Woodcock. Mechanising mondex with Z/Eves. *Formal Aspect of Computing*, 20(1):117–139, 2008.

[GBJI10] Gudmund Grov, Alan Bundy, Cliff B. Jones, and Andrew Ireland. The AI4FM approach for proof automation within formal methods – a Grand Challenge 6 "Dependable Systems Evolution" project. Grand Challenges in Computing Research (GCCR'10) – part of the ACM-BCS Visions of Computer Science 2010, April 2010.

[Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice Hall International, Englewood Cliffs, N.J., USA, second edition, 1993.

[Hen06] Alex Heneveld. *Using Features for Automated Problem Solving*. PhD thesis, School of Informatics, University of Edinburgh, Edinburgh, Scotland, February 2006.

[HJN94] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. *ACM Software Engineering News*, 19(3):75–81, July 1994.

[IGLB11] Andrew Ireland, Gudmund Grov, Maria Teresa Llano, and Michael Butler. Reasoned modelling critics: Turning failed proofs into modelling guidance. *Science of Computer Programming*, 2011.

[JDB11] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47:251–289, 2011.

[JGB10] Cliff B. Jones, Gudmund Grov, and Alan Bundy. Some facets of a strategy language for proofs. In *5th Automated Formal Methods workshop (AFM'10)*, July 2010. Also available as Edinburgh University, School of Informatics technical report EDI-INF-RR-1377.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[JKPB03] Mateja Jamnik, Manfred Kerber, Martin Pollet, and Christoph Benzmüller. Automatic learning of proof methods in proof planning. *Logic Journal of the IGPL*, 11(6):647–673, 2003.

[Jon79] C. B. Jones. Constructing a theory of a data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1979.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[JS90] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[KMM09] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *ACL2 Computer-Aided Reasoning: An Approach*. University of Austin Texas, 2009.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, May 2002.

[Saa97] Mark Saaltink. The Z/EVES system. In Jonathan Bowen, Michael Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer Berlin / Heidelberg, 1997.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Programming Research Group PRG126, Oxford University, 2000.

[Wen02] Markus M. Wenzel. *Isabelle/Isar - a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002.

[Woo08] Jim Woodcock, editor. *Formal Aspects of Computing*, volume 20:1. Springer-Verlag, January 2008.