COMPUTING SCIENCE

Proof Patterns for Formal Methods

Leo Freitas and Iain Whiteside

TECHNICAL REPORT SERIES

No. CS-TR-1399 November 2014

TECHNICAL REPORT SERIES

No. CS-TR-1399 November, 2014

Proof Patterns for Formal Methods

L. Freitas and I. Whiteside

Abstract

Design patterns are a highly successful technique in software engineering, giving a reusable 'best practice' solution to commonly occurring problems in software design. Taking inspiration, this paper introduces proof patterns, which aim to provide a common vocabulary for solving formal methods proof obligations by capturing and describing solutions to common patterns of proof, hence increasing effectiveness.

© 2013 Newcastle University.
Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

FREITAS, L., WHITESIDE, I.

Proof Patterns for Formal Methods
[By] L. Freitas and I. Whiteside
Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1399)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1399

Abstract

Design patterns are a highly successful technique in software engineering, giving a reusable 'best practice' solution to commonly occurring problems in software design. Taking inspiration, this paper introduces proof patterns, which aim to provide a common vocabulary for solving formal methods proof obligations by capturing and describing solutions to common patterns of proof, hence increasing effectiveness.

About the authors

Leo Freitas is a lecturer in Formal Methods working on the EPSRC-funded AI4FM project at Newcastle University. Leo received his PhD in 2005 from the University of York with a thesis on 'Model Checking Circus', which combined refinement-based programming techniques with model checking and theorem proving. Leo's expertise is on theorem proving systems (e.g. Isabelle, Z/EVES, ACL2, etc.) and formal modelling (e.g. Z, VDM, Event-B), with particular interest on models of industrial-scale. Leo has also contributed extensively to the Verified Software Initiative (VSTTE).

Iain J. Whiteside is a research associate on the EPSRC-funded AI4FM project at Newcastle University. Iain received his PhD in 2013 from the University of Edinburgh with a thesis 'Refactoring Proofs', which introduced a formal framework for reasoning about transformation on formal proof languages. In particular, his paper `Towards Formal Proof Script Refactoring' provided a mathematical understanding for refactoring in formal proof developments and won the best paper award at Conferences on Intelligent Computer Mathematics (CICM).
He has also worked at NASA Ames Research Center, developing a hierarchical structuring mechanism for Safety Cases in the AdvoCATE tool developed at Ames. He organised the 2013 CIAO workshop in Scotland, and the 5th Programming Languages for Mechanised Mathematical Systems workshop, a satellite workshop of CICM 2013 in Bath.

Suggested keywords

PROOF PATTERNS
FORMAL VERIFICATION
DESIGN PATTERNS

Proof Patterns for Formal Methods

Leo Freitas
Iain Whiteside

School of Computing Science, Newcastle University
{name.surname}@newcastle.ac.uk

Abstract

Design patterns are a highly successful technique in software engineering, giving a reusable 'best practice' solution to commonly occurring problems in software design. Taking inspiration, this paper introduces *proof patterns*, which aim to provide a common vocabulary for solving formal methods proof obligations by capturing and describing solutions to common patterns of proof, hence increasing effectiveness.

Contents

Chapter 1

Introduction

A key advantage of formal specification of software systems using mathematical models is a precise characterisation of the requirements that is amenable to scrutiny through deductive reasoning methods. Popular *formal methods* languages like Z, B, and VDM [1, 19, 30], share a similar methodology: a software system is modelled at an abstract level and refined stepwise to a more concrete representation by infusing the model with design decisions and concrete datatypes. In each level of 'refinement' a system state is usually described as an abstract datatype or record with an *invariant* attached. Operations of the system as described with preconditions and postconditions that may modify the system state. The correctness guarantees provided by 'formal methods' stem from the *proof obligations (POs)* that are generated for any given model. Within formal methods POs tend to have a predictable shape. Furthermore, the process of model design tends to exhibit predictable shapes. This repetition in the phrasing of theorems and in the solution to particular modelling problems suggests the possibility of repeated proofs. This repetitive notion of proof has been corroborated in practice from our personal experiences [10, 12, 13].

One of the main challenges for the widespread adoption of formal methods in industry is the expense, both in time and human expertise, of solving these proof obligations. For example, industrial partners using B claim each PO cost 38 euros! Our aim is to enable less experienced proof engineers to identify common situations and tackle them effectively by the use of proof patterns.

In Software Engineering, *design patterns* are a highly successful technique for providing solutions to frequently occurring problems in software design [14]. Patterns give a template description of how to solve a particular problem that can be used in many different situations. Patterns have evolved to become best practices that must be implemented and are easily recognised, whence providing a common vocabulary for describing solutions. Similarly, mathematicians do not often talk about the details of their proofs: they have a common parlance of high-level proof patterns (e.g. by induction, ϵ-δ proofs in analysis), that enables fellow mathematicians to understand and recreate proofs. It is our aim in this paper to introduce *proof patterns for formal methods*. Specifically, we hypothesise:

> *Similar to software engineering design patterns, proof patterns exists for formal methods proof obligations. Furthermore, under the right circumstances, they are applicable over multiple methods.*

This paper presents several proof patterns useful for formal methods proof obligations across methods. We describe these patterns similarly to software design patterns, and although we use a concrete example problem for explaining the patterns, we believe that they transfer across problems, and sometimes across provers. We provide examples of each pattern in action using a VDM model of a heap memory manager is used [20, Ch. 7]. We have formalised it in the Isabelle and Z/EVES theorem provers [24, 25]. We believe that too small an example is unlikely to clarify the issues with patterns in proof obligations, and that industrial examples do not fit in a paper (yet are amenable to our patterns).

In the next section, we briefly introduce software design patterns, formal methods, and our running example. Then, Section 3 describes our core proof patterns with examples of each in action. We introduce higher-level *methodological patterns* in Section 4 and further exemplify our proof patterns with a worked example on a feasibility proof obligation in Section 5. Finally, we conclude with related and future work in Section 7.

Chapter 2

Background

2.1 Patterns

Like with Design Patterns for software engineering [14], we see proof patterns as a combination of informal description, examples of use, and a set of attributes of discourse explaining the conditions for which the proof pattern may apply. A classic example of a design pattern is the *iterator pattern*.

We are trying to do for proof what design patterns did for software development: create a discourse of ideas and processes that might omit the specifics of "how", unless one is happy to look at the gory details (*i.e.* not only of large proofs, but of large amounts of failed proof attempts). We try to give precise and accountable details as much as possible, yet we are still some way from having an expressive enough declarative proof language to capture proof intent.

2.2 Heap problem

In this section, we provide an overview of a heap memory manager, modelled in VDM [20, Ch.7], that we use throughout this paper to exemplify our proof patterns. We have formalised and proved all proof obligations associated with the first two levels of refinement for this model in the Isabelle [24, 23] and the Z/EVES [25] theorem provers. A full description of this formalisation, including a detailed description of the translation between a VDM model and its representation in Isabelle, can be found in [10], and at `www.ai4fm.org/tr`. The model consists of two datatypes and two operations:

Loc: the type of a single adjacent memory location, represented as $\mathbb{N}$.

Free: the type of the heap as a collection of all free locations. At *level 0*, it is represented as the set $Free0 \triangleq Loc\text{-}\mathbf{set}$, whereas at *level 1*, it is represented as a map from start location to size that is *disj*oint and *sep*arate:
$Free1 = Loc \xrightarrow{m} \mathbb{N}_1$
inv $(f) \triangleq disj(f) \wedge sep(f)$

$$disj(f) \triangleq \forall l, l' \in \mathbf{dom}\, f \cdot l \neq l' \longrightarrow \mathit{locs\text{-}of}(l, f(l)) \cap \mathit{locs\text{-}of}(l', f(l')) = \phi$$
$$sep(f) \triangleq \forall l \in \mathbf{dom}\, f \cdot (l + f(l)) \notin \mathbf{dom}\, f$$

The invariant conditions ensure that the range of locations identified by any two map elements (defined as $\{l \ldots l + f(l)\text{-}1\}$ by *locs-of*) do not intersect (*disj*) and that contiguous memory regions are as large as possible (*sep*). That is, for any element l in the map, the location immediately to the right of its memory region ($l + f(l)$) is not the start location for another region: $l + f(l) \notin \mathbf{dom}\,(f)$. We write $F1\text{-}inv(f)$ to refer to this invariant of *Free*1.

NEW: takes a size and heap as input and returns a starting location for a contiguous chunk of memory of the appropriate size after updating the state.

DISPOSE: returns a contiguous chunk of memory back to the heap. This operation takes a start location and size as parameters, updating the state.

At level 0, these operations are defined as

$NEW0\ (s{:}\,\mathbb{N}_1)\ r{:}\,Loc$	$DISPOSE0\ (l{:}\,Loc, s{:}\,\mathbb{N}_1)$
ext wr f_0 : $Free0$	**ext wr** f_0 : $Free0$
pre $\exists l \in Loc \cdot \mathit{locs\text{-}of}(l, s) \subseteq f_0$	**pre** $\mathit{locs\text{-}of}(l, s) \cap f_0 = \{\,\}$
post $\mathit{locs\text{-}of}(r, s) \subseteq \overleftarrow{f_0}) \wedge$ $f_0 = \overleftarrow{f_0} - \mathit{locs\text{-}of}(r, s)$	**post** $f_0 = \overleftarrow{f_0} \cup \mathit{locs\text{-}of}(l, s)$

Set difference and set union characterise the removal and addition of elements to the heap. The precondition on $NEW0$ ensures that there is a contiguous region of the appropriate size, whereas $DISPOSE0$ ensures the range of locations being returned is not already free. At level 1, the NEW operation is:

$NEW1\ (s{:}\,\mathbb{N}_1)\ r{:}\,Loc$
ext wr f_1 : $Free1$
pre $\exists l \in \mathbf{dom}\, f_1 \cdot f_1(l) \geq s$
post $r \in \mathbf{dom}\, \overleftarrow{f_1} \wedge (\overleftarrow{f_1}(r) = s \wedge f_1 = \{r\} \lhd \overleftarrow{f_1} \vee$
$\overleftarrow{f_1}(r) > s \wedge f_1 = (\{r\} \lhd \overleftarrow{f_1}) \cup_m \{r + s \mapsto \overleftarrow{f_1}(r) - s\})$

$NEW1$ has two behaviours depending on whether a location of exactly the required size or strictly larger has been located, If the size matches, then that element is removed from the map; if the map element refers to a larger region, then the remaining locations in the region must be added back to the heap (hence the map union). The precondition captures both cases using $\geq$. We describe $DISPOSE1$ in Section 5 as part of a worked example of solving a feasibility proof obligation with proof patterns. The *retrieve* between these two levels of refinement is given by function $f_0 = locs(f_1)$: it generalises *locs-of* over the domain of f_1 using distributed union (i.e. $locs = \bigcup\{\mathit{locs\text{-}of}(x, f_1(x)) \mid x \in \mathbf{dom}\, f_1\}$).

Chapter 3

Proof Patterns

In the following sections we describe our proof patterns. Despite the specificity of the Heap example, these patterns apply in most problems of interest within formal methods POs. Wherever patterns are specific enough, we provide proof snippets in Isar: a human readable formal proof language for Isabelle [29].

3.1 Witnessing

One of the most important steps to solve feasibility (and reification) POs is finding appropriate witnesses for the outputs and updated state. The general form of these POs is $\forall \overleftarrow{\bar{\sigma}}, \bar{i} \cdot \mathit{pre\text{-}OP}(\overleftarrow{\bar{\sigma}}, \bar{i}) \longrightarrow \exists \bar{\sigma}, \bar{o} \cdot \mathit{post\text{-}OP}(\overleftarrow{\bar{\sigma}}, \bar{i}, \bar{\sigma}, \bar{o})$, where $\overleftarrow{\bar{\sigma}}$ and $\bar{o}$ are the initial state and inputs and $\bar{\sigma}$ and $\bar{o}$ represent a sequence of updated state variables and outputs. In general, finding a witness is a difficult task, but there are two common patterns that allow some of the existentials to be discharged easily and a third pattern to help the engineer "discover" the right unknown witness.

One point. Often the value of an updated state variable is given explicitly in the postcondition as an equality, $\sigma_i = t$, where t is an expression in terms of the initial state. This can be discharged by a generalised version of the "one-point' proof rule in [30, Sect. 4.2], which Z/EVES implements and could be encoded in Isabelle, though at present we discharge it manually. The rule avoids providing any explicit instantiation, regardless of variable order. If t is a complicated expression, one may wish to introduce and informative name $x' = t$ into the assumptions and use the one-point rule with x' instead, since t is substituted for σ_i everywhere it is used in the postcondition.

Existential precondition. A more subtle situation involves an existential precondition (e.g. $\mathit{pre\text{-}OP} \triangleq \exists x \cdot P\ x$). This often means that x, once eliminated, is 'supposed to be' mapped to a particular witness. Notice that inputs to preconditions can also be viewed as existentially quantified assumptions and are also suitable. This pattern occurs when a nondeterministic choice for the

postcondition is required and a value satisfying the precondition can be picked. This means that we can often match an "existential" precondition variable by trying to match the $P\ x$ from the precondition within the postconditions.

Dummy. When a witness guess is unknown, making progress in the proof serves to clarify the appropriate choice. To make progress, one should instantiate the existential quantifier with an arbitrary variable, then proceed. Once the goal is rewritten taking the witness into account, resulting subgoals can be analysed for evidence pointing to appropriate instantiations.

Example. In the heap case study, both one-point and existential precondition patterns occur in the feasibility PO for $NEW1$ and $DISPOSE1$. After performing case analysis, the first PO is as follows:

$$\forall f\ s \cdot F1\text{-}inv(f)\ \land\ (\exists l \in \mathbf{dom}\, f \cdot f(l) = s) \longrightarrow$$
$$(\exists f'\ r' \cdot F1\text{-}inv(f')\ \land\ r' \in \mathbf{dom}\, f\ \land\ f(r') = s\ \land\ f' = \{r\} \lhd f\ )$$

The witness for f' can be found using the one-point rule for $f' = \{r\} \lhd f$. This entails some existential introduction massaging, which Z/EVES does automatically and we encode in Isabelle. Notice we cannot use r in the witness as it is also being quantified. The witness for r is the l introduced by existential elimination on the precondition. The witnessing pattern reduces this feasibility proof obligation to showing that the invariant holds on the updated state as $F1\text{-}inv(\{l\} \lhd f)$, which is ready for invariant breakdown as described in the next Section.

3.2 Invariant breakdown

One often needs to show that the updated state preserves the invariant, as in the example above. When the updated state is defined in terms of the original state, then we move the invariant predicate towards the original state in order to use the assumption the original invariant held. This can be seen as a specialised form of the *rippling* proof plan for solving step cases of induction proofs [4]. Rather than delve into the details of rippling, we explain this pattern as an operation on invariant proofs.

The situation that triggers this pattern is as follows. We need to solve $inv(h(\sigma))$ where we know $inv(\sigma)$ holds and the updated state is $h(\sigma)$. As a simple example, imagine the state is a set of natural numbers X and the updated state is $f(X) \cap g(X)$, therefore giving us a proof obligation $inv(X) \longrightarrow inv(f(X) \cap g(X))$. To apply the invariant breakdown pattern, we aim to move the invariant predicate *closer* to the original invariant terms: distributing it over set intersection in this case. Furthermore, the subterms that contain the original state would be generalised to an arbitrary element. This means speculating a lemma: $P \longrightarrow inv(A) \longrightarrow inv(B) \longrightarrow inv(A \cap B)$, where P expresses side-conditions under which the lemma must hold. The application of this lemma leads to $inv(f(X))$ and $inv(g(X))$ as new subgoals, we need to apply the invariant breakdown pattern again until we get $inv(X)$ itself as assumptions to

the speculated lemma. While the invariant breakdown pattern does not solve the goal, it provides a set of lemmas that, if proven, will lead to a proof of the top-level goal. We call them weakening lemmas (see Section 3.3) and they are available to function symbols either unknown or with little automation.

The process of discovering such side-conditions P is non-trivial and requires expertise. It can also be helped, however, by model-checking and counter-example checking: take P as **true** and run a counterexample checker, for example. Another source of useful information is the definitions of the operators involved, in the case above $\{inv, f, g, \cap\}$. Together with counter-examples found, they expose the clues to the appropriate side-conditions, which might themselves be lemmas to be proved. In the worst case, when side-conditions are difficult to guess, one might need to create a new concept specific to the domain of the problem.

Example. In the heap, this pattern was used frequently, since the two components of the invariant (*sep*arateness and *disj*ointedness) needed to be proved for the updated state in both the NEW and DISPOSE operations. In $NEW1$, for example, we are required to show the following goal holds.

$Disjoint((\{r\} \lhd \overleftarrow{f_1}) \cup_m \{r + s \mapsto \overleftarrow{f_1}(r) - s\})$

where we have a single occurrence of the original state f_1 *as itself*, which gives us some indication of how to breakdown this formula by distributing *Disjoint* over map union ($\cup_m$). Generalising, we speculate a lemma:

$Q \longrightarrow Disjoint(f) \longrightarrow Disjoint(\{a \mapsto b\}) \longrightarrow Disjoint(f \cup_m \{a \mapsto b\})$

Ignoring side-conditions for the moment, this gives us the following subgoals:

$Disjoint(\{r\} \lhd \overleftarrow{f_1}) \qquad Disjoint(\{r + s \mapsto \overleftarrow{f_1}(r) - s\}))$

The second goal does not contain the original state, and can be solved trivially. For the first goal, we need to repeat the invariant breakdown process on domain filtering ($\lhd$). This allows us to solve the goal using the assumption.

3.3 Weakening lemmas

When one does not have enough information about function symbols appearing in invariant breakdown, we need weakening lemmas relating these symbols.

For instance, POs over complex state often include records and data structures the prover knows little about (e.g. a map from a record to a list). Naively dealing with the presence of these (novel use of) symbols often leads to either polluted (and repetitive) proofs, or to overly specific lemmas. Instead, we need lemmas that weaken specific parts of the goal (for backward reasoning) or specific parts of the hypothesis (for forward reasoning). This breaks down the task to manageable pieces, up to the point where the prover has automation for function symbols involved, as in the example above involving $inv(\_ \cap \_)$.

When discharging proof obligations for the heap we came across surprising points of failure. The refinement and feasibility POs for *NEW* and *DISPOSE*

motivated the creation of lemmas for both Z/EVES and Isabelle, which are fully documented in [10].

Isabelle proofs. From the pattern for the feasibility proof, we provide weakening lemmas to enable automation on our given witnesses This unpicking of the various parts of the feasibility proof obligation leads to the suggestion of lemma shapes up to the point where available lemmas apply. Once these linking (weakening) lemmas are in place, Isabelle knows enough about involved operators and can automatically discharge them. Bridging this gap is where expert input is needed. For example, to represent VDM maps in Isabelle we use ($Loc \rightharpoonup \mathbb{N}$), which is a total function with an optional range type. Isabelle has useful operators like map update ($\dagger$), yet not map union ($\cup_m$), which we define as map update over maps with disjoint domains. Map union features in proofs for $NEW1$ and $DISPOSE1$, and we needed lemmas linking the new operator to a representation known to Isabelle, namely map update. Thus, while performing *invariant breakdown*, we identify the need for weakening lemmas about distributing VDM operators over invariant subparts. Similar lemmas for other function symbols were also added, where different side-conditions determine where such weakening lemmas can be used. In total we have 51 of them for Isabelle.

Z/EVES proofs. Arguably, given Z already contains a mathematical toolkit akin to VDM's, it is easier to represent a VDM model in Z/EVES. This means, many of the weakening lemmas we need are already available. The lesson from this though is that weakening lemmas over known (or reused) function symbols do indeed transfer across problems, and that is our experience. Some of the Isabelle weakening lemma were informed by previous experience with Z/EVES.

We declare weakening lemmas in Z/EVES as a rewrite rules, which the prover uses automatically during simplification and is akin to Isabelle's simp attribute on lemmas. These lemmas are not quite solving, but distilling the problem through the proof engineering process described by our proof patterns. These kinds of lemmas are not usually transferable across problems, yet the general principles/patterns behind them are.

3.4 Type bridging and zooming

When discharging weakening lemmas and/or discovering side-conditions of invariant breakdown, one often needs to add explicit (**novel, if obvious**) information about type relationships and their layers of abstraction/representation. These lemmas establish algebraic properties between new user-defined operators, and known (set theory, say) operators.

Provers have preferred directions for reasoning, be that left-right simplification, or a "waterfall" [22] involving generalisation and simplification. We call "type bridging" all those auxiliary lemmas necessary for weakening the goal (in backward proof) towards *true* by substituting it by "simpler" goals up to the point they meet the available hypothesis. For example, when a conditional rule

fails to match, it is often because its side-condition could not be discharged. Often these conditions are just type checking like parameters are within the involved function symbols declared types. This gives rise to a set of specialised type-inference lemmas for the involved expressions.

The mathematical toolkits of Z, Event-B, and VDM are defined in terms of maps, sequences, sets, and relations. User-defined functions are often given in terms of operators with higher automation, in which case the appropriate expansion lemma to the function symbol with most automation is needed. In methods such as Z and VDM, the notion of records is ubiquitous yet goals involving records do not require the prover to know about all the record structures, hence specific lemmas exposing a record's properties are often needed to streamline proofs (*i.e.* in [12], such record slicing reduced a 45-page long cluttered goal into 16-chunks of related intent about a page each).

This type bridging, where the lemma is there to help bridge notions between operators of interest (*i.e.* user-defined and set theory in this case) enhances proof automation. Similarly, when moving between layers of abstraction/representation, the right level of definition expansion needs to be taken into account. That is, we need to instruct the prover what "zoom" to use, and we do not want to expand all definitions to sets or predicates, but rather keep definitions at different "zoom" levels, adding lemmas between levels as needed.

Such type judgements can also work in forward proof by strengthening hypothesis of interest. For instance, given a goal involving an injective map inverse ($f\tilde{}(f(x))$) and an assumption $x \in \mathbf{dom}\, f$, we could extend the hypothesis to say that $f\tilde{}(f(x)) = x$. This usually makes the prover substitute the goal with the simpler RHS involving x. The amount and shape of these auxiliary lemmas are determined by the direction a prover takes, as well as by the amount of previously available information for the given operators. Finally, certain invariants are mathematically sensible (*i.e.* a sequence of size up to 10, $s \in T^* \land card(s) \leq 10$), yet hard to use in proof because of tricky operators like cardinality ($card(s)$) that involve bijective functions. Equivalent versions of the same invariant using just set theory (i.e. sequence indices range from 1 up to 10, $\mathbf{inds}\, s \subseteq 1, \ldots, 10$) can be proved as type bridging lemmas, hence providing better automation.

Example. The Isabelle proof for the heap example we added a (congruence) lemma that required no extra side-conditions, hence directly simplifying the goal by removing one of the operators (*i.e.* $sep(x \lhd f) = sep(f)$). It states that subtracting from the map's domain preserve separation. The presence of the operator to absorb on the LHS tells the prover our preference for the RHS expression as a result. Moreover, the free variables in weakening lemmas for *sep* needed for the feasibility of $NEW1$ were reused in $DISPOSE1$, hence making the feasibility proof script itself much like the one for $NEW1$ (*i.e.* their common strategy being reused modulo key lemmas discovered).

3.5 Retrieve state update

When refining a model, a *retrieve* relation maps concrete to abstract state representations. A key PO is that if the postcondition holds at the concrete level, then it holds in the abstract. This proof obligation is called *narrow postcondition* in VDM (correctness in Z). A common feature in refinement is a *type jump* between the abstract and the concrete levels. For example, the heap is represented as a set of locations at level 0 and refined to a partial map in level 1.

In these situations where the updated state at the concrete level is described using an equality $\sigma = f(\overleftarrow{\sigma})$ for some f, or when we have a functional retrieve, we can apply a pattern called *retrieve state update* to conjecture lemmas that map across the type jump. To prove the postcondition we can use one-point witnessing under the retrieve to solve $retr(\sigma) = g(retr(\overleftarrow{\sigma}))$ for some function g on the original state, where $retr(con) = abs$ is a function from the concrete to the abstract state. After the state update with the equality above, this is really: $retr(f(\overleftarrow{\sigma})) = g(retr(\overleftarrow{\sigma}))$. In this situation, we have three pieces of information: a) the structure of the retrieve function mapping concrete and abstract; b) a set of operators at the concrete level (used in f); and, c) a set of operators at the abstract level (used in *retr* and g). To solve this goal, we must first translate the operators in f to those of g by distributing the retrieve function. For example, the narrow postcondition PO for the NEW operation on the heap is:

$$retr(\{r\} \lhd \overleftarrow{f}) = locs(\overleftarrow{f}) - locs\text{-}of(r, s)$$

The 'zoom' level for the abstract state is that of sets and the level of the concrete is maps. The application of this pattern suggests distributing *retr* over domain filtering, which suggests a possible lemma of the form

$$P \longrightarrow retr(\{r\} \lhd \overleftarrow{f}) = Q(retr(\overleftarrow{f}), \{r\})$$

where P expresses side-conditions and Q is some undetermined operation on sets. At this point the retrieve function is operating solely on the state, so we stop applying the pattern. Comparison with the right hand side of our goal makes clear what Q should be, but this is not always so straightforward. Thus, just like invariant breakdown, the proof pattern can help suggest an attack, but it requires input from the proof engineer.

3.6 Hidden Case Analysis

Hidden case analysis is the insertion of a lemma of the form $P \vee \neg P$ to the hypotheses and the subsequent breakdown of the proof obligation into the case where P holds and the case where $\neg P$ holds. This pattern is common to mathematics as well. It is often used in situations where it is not clear what information the precondition provides to prove the postcondition holds, despite it being available, if hidden. A few examples of where this occurs are:

- Certain predicate calculus patterns in either pre/postconditions lend themselves to hidden cases analysis. Conditional postconditions ($P \longrightarrow Q$) in-

dicate a case split on the condition. Disjunctive postconditions often are associated with hidden disjuncts of the precondition (e.g. as in $NEW1$).

- Non-linear arithmetic operators occur in the precondition, such as $\geq$, and also when negation and non-linear arithmetic are combined.
- Sets, sequences, and maps in preconditions might require case analysis for emptiness; and so on.

For example, the disjunction in the postcondition for the $NEW1$ operations

$$(\overleftarrow{f_1}(r) = s \wedge f_1 = \{r\} \lhd \overleftarrow{f_1}) \vee (\overleftarrow{f_1}(r) > s \wedge f_1 = (\{r\} \lhd \overleftarrow{f_1}) \cup \{r + s \mapsto \overleftarrow{f_1}(r) - s\})$$

suggests that a case analysis must be performed on the preconditions. In this case, introducing $f_1(l) = s \vee f_1(l) \neq s$ allows us to derive from the precondition $(\exists l \in \mathbf{dom}\, f_1 \cdot f_1(l) \geq s)$ that $f_1(l) > s \vee f_1(l) = s$. Each case in the precondition is explicit as a goal in each side of the disjunction, although the disjunct with the precondition was hidden by $\geq$.

3.7 Shaping

In formal methods, proof obligations tend to be large with lots of information present in the goal and assumptions. This can often obscure the overall structure of a goal. In the shaping proof pattern, we utilise some of this information to simplify the goal before applying an important lemma or applying another proof pattern, such as invariant breakdown. A shaping pattern consists of a set of shaping lemmas, which are equalities between a sub term of the goal and a simpler representation, and a set of targeted rewrites that simplify the goal. The benefit of using this proof pattern before applying an important weakening lemma is that it will considerably simplify the resulting subgoals. In mathematics, this often occurs when trying to get rid of some difficult operator like square root, so one squares both sides of an equation, say.

In the heap example, shaping was used frequently to simplify some of the details of the $DISPOSE1$ postcondition in the feasibility and refinement proofs, where prior case analysis had provided further information about the structure. We give an example of shaping (and the case analysis that triggers it) next.

Chapter 4

Methodological Patterns

The top-level POs for a model, such as operation feasibility, tend to have similar structure that can be exploited to increase proof effectiveness. We call such similarities *methodological patterns*. In this section we identify methodological patterns that are independent of specification method,namely the top-level goal of operation feasibility and data refinement proof obligations of B, Z, and VDM. The strategy to handle such goals is mostly always the same. We use VDM POs in our running example.

4.1 Feasibility (or satisfiability) obligations

Example from the heap. The full feasibility proof obligation from the NEW operation of the heap problem is as follows:

$$\begin{array}{l} \forall \cdot f\ s.\ \mathit{F1\text{-}inv}\ f \land \mathit{nat1}\ s \land (\exists \cdot l \in \mathit{dom}\ f.\ s \leq \mathit{the}\ (f\ l)) \longrightarrow \\ \quad (\exists \cdot f'\ r'.\ \ \mathit{F1\text{-}inv}\ f' \ \land (r' \in \mathit{dom}\ f \land \mathit{the}\ (f\ r') = s \land f' = \{r'\} \lhd\!\text{-}\ f \lor \\ \qquad r' \in \mathit{dom}\ f \land s < \mathit{the}\ (f\ r') \land f' = \{r'\} \lhd\!\text{-}\ f \cup m\ [r' + s \mapsto \mathit{the}\ (f\ r') \text{ - } s]) \ ) \end{array}$$

where f is the initial state representing the heap, s is the input, representing the amount of space required. The existentially quantified f' is the updated state, and r represents the output location (marking the start of the allocated space). The preconditions consist of an invariant on f, a simple typing precondition on the input, and an existential precondition. The postcondition is a disjunction of the two possible cases discussed in Section X.

The first step of a feasibility pattern is to perform any necessary transformations and structural introduction and elimination rules (so-called 'safe' rules') to fully expose the goal we wish to solve. The combination of disjunction in the conclusions and the existential precondition hints at a case split (with $\leq$), which results in two goals. We apply the feasibility pattern to each. We'll focus on the $>$ case in this example. The goal we need to solve is now:

$$\frac{\begin{array}{l} F1\text{-}inv\ f \\ nat1\ s \\ l \in dom\ f \\ the(f\ l) > s \end{array}}{\begin{array}{l} \exists \cdot f'\ r'.\ (r' \in dom\ f \wedge the\ (f\ r') = s \wedge f' = \{r'\} \lhd\!\text{-}\ f \vee \\ r' \in dom\ f \wedge s < the\ (f\ r') \wedge f' = \{r'\} \lhd\!\text{-}\ f \cup m\ [r' + s \mapsto the\ (f\ r') \wedge F1\text{-}inv\ f' \end{array}}$$

(Note that we have instantiated the existential precondition, thus l has joined our set of fixed variables)

At this point, we wish to get rid of the disjunctions and choose one. Technically, we should do a search on all possibilities, but we can always distribute the existentials to create a copy for each disjunction. Clearly in this case, we choose the RHS:

$$\exists \cdot f'\ r'.\ r' \in dom\ f \wedge s < the\ (f\ r') \wedge f' = \{r'\} \lhd\!\text{-}\ f \cup m\ [r' + s \mapsto the\ (f\ r') - s]) \wedge F1\text{-}inv\ f')$$

Can we? Should we? Tighten this up

It is at this point we can being the feasibility witnessing pattern. Here we only have two variables to instantiate and decide on an ordering. We spot that f' is a single point , but it refers to r'. Thus, we need to instantiate r' before (this requires an existential swap). For r', we note that the postconditions involving r' are contained within the assumptions, only referring to l. This confirms the intuition that, since l was an existential precondition it represents an output. We can then instantiate r' as l and $f' = \{r'\} \lhd\!\text{-}\ f \cup m\ [r' + s \mapsto the\ (f\ r') - s]$.

All of the postconditions can then be discharged by either reflexivity or by assumption (as a result of our feasibility witness choices). We are left with the invariant to solve:

$$\frac{F1\text{-}inv\ f}{F1\text{-}inv(\{r'\} \lhd\!\text{-}\ f \cup m\ [r' + s \mapsto the\ (f\ r') - s])}$$

Or rather, since the $F1$-inv is a conjunction of individual parts of an invariant, we have a goal for each. Let's take $Disjoint$, for example:

$$\frac{Disjoint\ f}{Disjoint(\{r'\} \lhd\!\text{-}\ f \cup m\ [r' + s \mapsto the\ (f\ r') - s])}$$

To this, we apply the invariant breakdown strategy. We see that the top symbol in the updated state is map union $\cup m$ and thus speculate a lemma:

$Disjoint(\{r'\} \lhd\!\text{-}\ f\ ) \longrightarrow P \longrightarrow Disjoint(\{r'\} \lhd\!\text{-}\ f \cup m\ [r' + s \mapsto the\ (f\ r') - s])$

to break down the goal using the top level operator. We write P to represent to unknown conditions that an expert must supply[1]. This breaks down our goal to

$$\frac{Disjoint\ f}{Disjoint(\{r'\} \lhd\!\text{-}\ f\ )}$$

which is represented as a further lemma:

$Disjoint(f) \longrightarrow P \longrightarrow \{r'\} \lhd\!\text{-}\ f\ )$

which breaks down the goal so that we can use the invariant assumption. Thus, straightforward application of proof patterns has reduced the PO to proving two lemmas (both of which actually involve subtle side-conditions) and can be generalised.

[1] Though, of course, we could always supply the full suite of PO conditions

FIX ME: Rewrite the invar breakdown to generalise the side of an operator that contains the assumption! That is, so that it would conjecture the general disjoint-unionm-singleton lemma

4.2 Refinement obligations

Another class of proof obligations are those associated with data reification, where data refinement establishes a link between different levels of design decisions, from abstract to concrete representations. Refinement goals are well structured and method independent. The pattern to handle them is similar to the one for feasibility proofs. Its proof obligation may vary depending on the kind of refinement done, yet the most common situation is forward simulation. They are known as applicability and correctness (in Z), or widen-pre and narrow-post in VDM, and are proved for every operation involved. The widen-pre PO state that under a retrieve relation (or function) linking the abstract and concrete state representations, the concrete precondition is at least that same or weaker than the abstract precondition.

$$\forall \overleftarrow{\sigma_a}, \overleftarrow{\sigma_c} \cdot \mathit{pre\text{-}AOP}(\overleftarrow{\sigma_a}) \land R(\overleftarrow{\sigma_a}, \overleftarrow{\sigma_c}) \;\Rightarrow\; \mathit{pre\text{-}COP}(\overleftarrow{\sigma_c})$$

Conversely, the narrow-post PO state that it is possible to find a corresponding abstract (after) state satisfying the abstract postcondition under the retrieve, providing the concrete postcondition and abstract precondition holds under the retrieve.

$$\forall \overleftarrow{\sigma_a}, \overleftarrow{\sigma_c}, \sigma_c \cdot \mathit{pre\text{-}AOP}(\overleftarrow{\sigma_a}) \land R(\overleftarrow{\sigma_a}, \overleftarrow{\sigma_c}) \land \mathit{post\text{-}COP}(\overleftarrow{\sigma_c}, \sigma_c)$$
$$\Rightarrow\; \exists \sigma_a \cdot R(\sigma_a, \sigma_c) \land \mathit{post\text{-}AOP}(\overleftarrow{\sigma_a}, \sigma_a)$$

The most difficult part in this category of proof obligation is to find an appropriate retrieve linking the two representations, such that all operations satisfy the two POs above. In practice, it is often necessary to adjust the retrieve and revisit these two proofs for every operation involved.

The proof pattern to use in this situation is much like the one for feasibility proofs. The widen-pre PO does not require witnessing, but invariant breakdown now under the retrieve. The narrow-post PO is much like feasibility proof: first find the appropriate abstract state witness to satisfy the abstract postcondition given the concrete postcondition update. This will involve invariant-breakdown on both POs for both abstract and concrete states.

A common case for this situation is when you have a retrieve function relating both states, which in VDM is known as the adequacy PO. It states that for every concrete state representation there exists only one abstract representation under the retrieve.

$$\forall \overleftarrow{\sigma_c} \cdot \exists \overleftarrow{\sigma}_a \cdot \overleftarrow{\sigma_a} = R(\overleftarrow{\sigma_c})$$

When the retrieve is not defined as a function but a relation, yet only one relating value exists, unique existential quantifier ($\exists! \cdot$) is used. Proving adequacy is useful because it establishes the abstract state witness to be used for the narrow-post PO for every operation, hence simplifying the refinement proof considerably. It is usually proved by induction over the types involved. In Z,

this is known as functional refinement. Z also have a similar set of proof obligations for backward simulation, where resolution of nondeterminism is postponed and it is as if the abstract state is simulating the concrete state by anticipating its actions. The proof pattern to use is the same: witnessing and invariant-breakdown.

Chapter 5

Patterns in the heap

Rather than describe these patterns, we give a worked example of how the patterns described in the previous section can be composed to help solve the feasibility proof obligation for the *DISPOSE* operation at level 1. In a companion technical report, we provide a detailed presentation of methodological patterns [11]. The VDM operation is as follows:

$DISPOSE1\ (d\colon Loc, s\colon \mathbb{N}_1)$

ext wr $f\ :\ Free1$

pre $is\text{-}disj(locs\text{-}of(d,s), locs(f))$

post $\exists below, above, ext \in Loc \xrightarrow{m} \mathbb{N}_1 \cdot$
$below = \{l \mid l \in \mathbf{dom}\, f \wedge l + \overleftarrow{f}\,(l) = d\} \lhd f \wedge$
$above = \{l \mid l \in \mathbf{dom}\, f \wedge l = d + s\} \lhd f \wedge ext = above \cup_m below \cup_m \{d \mapsto s\} \wedge$
$f = (\mathbf{dom}\, below \cup \mathbf{dom}\, above \lhd \overleftarrow{f}\,) \cup_m \{min\text{-}loc(ext) \mapsto sum\text{-}size(ext)\}$

The inputs d and s are the start location and size of region to add back to the heap. The precondition is similar to level 0: this time using *locs* to construct the set of all free locations from the heap map.

The complexity in $DISPOSE1$ arises from the fact that the memory region being added back to the heap may adjoin zero, one, or two other regions already in the heap. Thus, to preserve the *sep* part of the invariant (e.g. memory regions should be as large as possible), we must join them together. The map *above* will adjoin the end of the region being added; *below* defines the map of elements adjoining from the start. The *ext*ended map then consists of *above*, *below*, and the disposed region in the middle. Updating the state is then a case of removing above and below (using domain filtering) and adding a region that corresponds to the minimum starting location and the sum of the sizes of the elements of *ext*. To illustrate, for a heap $f = \{0 \mapsto 4, 8 \mapsto 3\}$, $DISPOSE1(4,4)$ would result in the updated $f = \{0 \mapsto 11\}$. This is because $above = \{8 \mapsto 3\}$, $below = \{0 \mapsto 4\}$, and $ext = \{0 \mapsto 4, 4 \mapsto 4, 8 \mapsto 3\}$.

Proof of *DISPOSE*1 feasibility by patterns.

Step 1: representation transformation. This proof pattern can be used optionally at the start of any proof in order to make a representational change to simplify the proof. All that is required is a lemma which equates both POs. In the case of *DISPOSE*1, we gave explicit definitions for *above*, *below*, and *ext*, which makes it easier to deal with existential quantifiers, and we prove this alternative definition equal to the original. It is worth mentioning one ought not change the model for the sake of proof alone, as this often impairs model clarity.

Step 2: safe decomposition. Not a formal methods proof pattern per se, but *safe decomposition* is used as a standard technique in automated reasoning to break down fixed variables, hypotheses and conclusions of a goal. We extend this technique in our Isabelle development to unfold the definitions of *pre-DIS*1 and *post-DIS*1-*ALT* a single zoom level. The result, given as a declarative Isabelle/Isar proof script is as follows:

theorem *DIS1-feas*: $(\forall \cdot\ f\ d\ s\ .\ \textit{pre-DIS1}\ f\ d\ s \longrightarrow\ (\exists \cdot\ f'\ .\ \textit{post-DIS1}\ f\ d\ s\ f'))$
proof (*subst dispose-feas-transform*) — The transformation step
 show $(\forall \cdot\ f\ d\ s\ .\ \textit{pre-DIS1}\ f\ d\ s \longrightarrow (\exists \cdot\ f'\ .\ \textit{post-DIS1-ALT}\ f\ d\ s\ f'))$
 unfolding *pre-DIS1-def post-DIS1-ALT-def*
 proof (*intro allI impI*, *elim conjE*) — Safe decomposition
  fix *f d s* **assume** *inv*: *F1-inv f* **and** *pre*: $(\textit{locs-of}\ d\ s) \cap (\textit{locs}\ f) = \{\}$
  show $\exists \cdot f'.\ f' = (\textit{dom}\ (\textit{below}\ f\ d) \cup \textit{dom}\ (\textit{above}\ f\ d\ s))\ \text{-}\!\lhd f \cup m$
        $[\textit{min-loc}\ (\textit{ext}\ f\ d\ s) \mapsto \textit{sum-size}\ (\textit{ext}\ f\ d\ s)] \wedge \textit{F1-inv}\ f'$
  gap — The gap represents the area of the proof still to solve
 qed
qed

Step 3: witnessing. In this case, witnessing is straightforward as we have a one-point existential. The resulting proof script replaces the previous **gap**.

show $\exists \cdot f'.\ f' = (\textit{dom}\ (\textit{below}\ f\ d) \cup \textit{dom}\ (\textit{above}\ f\ d\ s))\ \text{-}\!\lhd f \cup m$
      $[\textit{min-loc}\ (\textit{ext}\ f\ d\ s) \mapsto \textit{sum-size}\ (\textit{ext}\ f\ d\ s)] \wedge \textit{F1-inv}\ f'$
proof(*rule exI*, *rule conjI*, *rule refl*) — Single-point witnessing with exI
 show $\textit{F1-inv}\ ((\textit{dom}\ (\textit{below}\ f\ d) \cup \textit{dom}\ (\textit{above}\ f\ d\ s))\ \text{-}\!\lhd f \cup m\ \ldots)$
 gap
qed

Step 4: hidden case analysis. At this point, we could apply the invariant breakdown pattern, but an expert proof engineer notes that the definition of *above* and *below* means it is either empty or a singleton map. Thus, the next proof step is to apply case analysis on both above and below, resulting in four separate cases to be solved. We focus on the first, where $\textit{above} = \textit{below} = \phi$.

Step 5: shaping. We again postpone invariant breakdown and perform the shaping pattern since we have added the hidden case analysis information

above to the hypotheses. We now know, for example, that filtering the (empty) domains of *above* and *below* from f will have no effect. The result of case analysis and shaping leaves another proof gap as:

proof (*cases below f d = empty*, *cases above f d s = empty*) — case analysis
assume *below-empty*: *below f d = empty* **and** *above-empty*: *above f d s= empty*
have *ab-shape*: (*dom* (*below f d*) ∪ *dom* (*above f d s*)) -⊲ *f*= *f* **gap**
have *min-loc-shape*: *min-loc* (*ext f d s*) = *d* **gap**
have *sum-size-shape*: *sum-size* (*ext f d s*) = *s* **gap**
show *?thesis*
proof (*subst ab-shape*, *subst min-loc-shape*, *subst sum-size-shape*)
show *F1-inv* (*f* ∪*m* [*d* ↦ *s*])
gap
qed
. . . — the other three cases are not shown
qed

Step 6: zooming and decomposition. We wish to attack each part of the invariant (*sep* and *disj*) independently, so we unfold definitions and decompose the conjunction accordingly.

Step 7: invariant breakdown. We are now in a position to apply the invariant breakdown proof pattern. We describe the application of the pattern for *disj*, but the *sep* part is similar. Recall, from above, that we have $F1\text{-}inv(f)$ as an assumption (and thus also *disj f*). The aim of invariant breakdown is to speculate lemmas that expose the hypothesis in our conclusion. In this case, we have the original state on the left hand side of the map union operator, but not the right. This helps us to speculate a lemma: $P \rightarrow disj(f) \rightarrow disj(f \cup m\ [d \mapsto s])$ where P represents unknown side-conditions. In the case of *disj*, the important condition is (*locs-of d s*) ∩(*locs f*)={}, which is exactly the precondition for *DISPOSE*1. The (partial) proof script is:

show *disj* (*f* ∪*m* [*d* ↦ *s*])
proof (*rule unionm-singleton-disj*) — The speculated lemma
show *sep f* **using** *inv* **by** *assumption* — Use of hypothesis
show (*locs-of d s*) ∩ (*locs f*) ={} **by** *(rule pre)* — Precondition as side-condition
. . . — Additional side-conditions
qed

The use of invariant breakdown does not solve the invariant subgoals, but it directs the proof engineer to the appropriate lemma structures to speculate.

Step 8: weakening and type bridging. Because of the repeated nature of some formal methods POs, weakening lemmas speculated by invariant breakdown are often reusable. The lemma *unionm-singleton-disj* is used in each of the other three cases introduced by hidden case analysis.

Lemma inference is quite hard in general, yet within the proof process presented, weakening and type bridging lemmas that will make proof progress are

easier to identify. For example, in the proof for the other cases in the *DISPOSE*1 feasibility PO, because of the presence of $inv(f \cup_m g)$, we need a side-condition that the map domains are disjoint ($\mathbf{dom}\, f \cap \mathbf{dom}\, g = \{\}$). In context this is given by the rather specific lemma:

lemma *l-dispose1-munion-disjoint*:
dom (((*dom* (*dispose1-below f1 d1*) ∪ *dom* (*dispose1-above f1 d1 s1*)) -◁ *f1*) ∩
dom [*min-loc* (*dispose1-ext f1 d1 s1*) ↦ *sum-size* (*dispose1-ext f1 d1 s1*)] = {}

This lemma is an instantiation of a side-condition that features in all subgoals when breaking down the invariant for *DISPOSE*1, hence is the key enabler of progress for the invariant breakdown of that feasibility proof.

Chapter 6

Evaluation

Before one can get to the nub of the problem within industrial-scale proof obligations, which almost always involve large formulae (i.e. tens of pages long) and multiple (i.e. over 100) variables, we claim it is fundamental to have in place a considerable amount of machinery to enable automation to an acceptable level. Proof engineering is essential for scalability: it takes a good amount of unrelated proof effort to enable one to tackle the actual proof obligations of interest. Lemmas are useful whenever one needs to either: decompose a complex problem; fine-tune the theorem prover's rewriting abilities to given goals; generalise a solution of some related (usually more abstract) problem; and to provide alternative solutions/encodings of the same data structure/algorithm being modelled; *etc.*

In our experiment we test our hypothesis by having the same proof task performed independently by three different people with three different backgrounds (i.e. formal methods proof expert, Isabelle proof expert, MSc student), in two different provers (i.e. Isabelle/HOL and Z/EVES), and encoded in two different methods (i.e. VDM and Z) on medium size refinement problem (i.e. the Heap). The rationale for this experimental setup was to check a few points:

- how does proof expertise differ and why?
- how much can be handled by a novice industrial proof engineer (i.e. MSc graduate)?
- what are common factors, and are there crucial patterns used by all parties?

We did a rough-and-ready analysis the proof traces and scripts of the Isabelle development (using Perl) looking for commonalities and differences. On the expert proof engineer development, our new lemmas on VDM maps in Isabelle were the ones with highest reuse rate (at 22%), with other available Isabelle library lemmas reuse being quite high too (at 38%). On the Isabelle expert, the ratio was slightly different at 16% and 65% respectively. The effort on PO-specific weakening lemmas and type bridges was comparable at 23% and 17%

for each expert. This indicates that a considerable amount of effort (around 20% for both experts) was related to setting up VDM map operators and lemmas in Isabelle, whereas around the same effort was needed on the actual POs. Arguably, the VDM lemmas are reusable across problems, hence the patterns described for the Heap problem do transfer across problems (in VDM at least). The encoding in Z/EVES was relatively straightforward, as there were no issues with undefinedness and the Z mathematical toolkit is quite similar to VDM's. This part of the experiment was useful, however, in early detection of possible proof-difficulty in the model, which only appeared much latter in the Isabelle development.

The MSc student Isabelle proofs [26] were also an interesting source for our tools and methods. She had no experience with either VDM or Isabelle, or indeed formal methods. Yet, after a few months of (mostly self-)training, she managed to finish the feasibility proofs for $NEW0/1$, and got stuck on $DISPOSE1$, which is rather difficult. The interesting outcome was, having been exposed to the proof patterns we described after Isabelle training, she also came by similar lemmas from our VDM Isabelle library.

Proof effort. The complete proof effort, its description, theory files, proof process meta-data, etc can be found in our AI$_4$FM archive in `http://www.ai4fm.org/tr`. In one of our (3) Isabelle developments, we defined (and used) 105 lemmas involving various operators, of which 51 were weakening lemmas. The overall development had *XXX* goals and *ZZZ* proof steps. It involved three versions of the same model, changed due to identified mistakes on the original version [20], and some of our own versions. The (single) Z/EVES effort was on a preliminary version of the model, and served to identify key relationships between the various function symbols involved and identify necessary weakening lemmas. These identified lemmas were useful for both Z/EVES and Isabelle proofs.

Throughout this process, we use our AI$_4$FM tools to capture the proof process data for all three users doing proof in both Isabelle and Z/EVES. This trove of meta-proof data is yet to be analysed and is quite large (e.g. over 250 GB). It contains all proof attempts alongside model diff-histories per attempt. Our plan is to investigate this data to identify how experts / novices react and recover from (proof-)failure, as well as whether there are clusters of useful information and proof intent. Eventually, we hope to use this data to have a statistically (machine-learned) ground notion of proof patterns usefulness, instead of an empirical one based on experience and suggestion.

Chapter 7

Related work and conclusions

This paper introduced the concept *proof patterns in formal methods* and described several patterns of proof that commonly occur. We tested this hypothesis by an experiment: the proof of a VDM heap memory manager [20] Just as in software design patterns [14], proof patterns are informal and are described generically, without reference to an individual problem. We also describe the composition of a set of patterns in a worked example of a feasibility proof obligation. We believe that a small collection of proof patterns is all that you need to increase the automation of most formal methods proof obligations. This **will not** eventually remove the burden of proof, yet we believe proof patterns offer good support in de-skilling the process, as well as increasing proof effort reuse.

Before getting to the nub of the problem within industrial-scale proof obligations, which almost always involve large formulae (i.e. tens of pages long) and multiple (i.e. over 100) variables, we claim it is fundamental to have in place a considerable amount of machinery to enable automation to an acceptable level. Proof engineering is essential for scalability: it takes a good amount of unrelated proof effort to enable one to tackle the actual proof obligations of interest. Lemmas are useful whenever one needs to either: decompose a complex problem; fine-tune the theorem prover's rewriting abilities to given goals; generalise a solution of some related (usually more abstract) problem; and to provide alternative solutions/encodings of the same data structure/algorithm being modelled; *etc.*

Related work. The term design pattern originated in architecture [2], but is most widely known in software design [14]. Some languages, such as Java even have built in support from patterns, such as the *iterator* pattern. In software engineering, there is a wealth of research in design patterns, from architectural patterns to specific patterns for user interfaces. In [8], Buschmann introduces architectural patterns that are capable of describing large-scale software sys-

tems, such as model-view controller, which separates the representation of data (the model) and the user's view of it. Architectural patterns are similar to our methodological patterns, which we describe as a composition of proof patterns to help solve a top-level proof obligation. We describe some closely related work in formal proof, but Buschmann et al. has related work on design patterns [7].

While we do not know of any specific research on proof patterns in formal methods, it has been noted in the mathematical community that pencil-and-paper proofs often follow specific patterns, such as proof by contradiction, by induction, etc. These ideas have found their way into the domain of automated theorem proving. Bundy's proof plans were an early attempt at capture patterns of inductive proof [5]. Proof plans have been implemented many times, most recently in Isaplanner [9]. We would like to investigate if the planning language in Isaplanner is expressive enough to formalise some of our proof patterns. In [21], we start this process by describing a language to capture meta-proof information. Also within AI4FM, colleagues have developed a graphical rewriting language [15] that will hopefully be amenable to proof pattern recognition. As well as the static proof patterns described here, it may be possible to learn patterns from a corpus of proofs [17, 18].

Future work. In the heap example, we also used AI4FM tools (under-development) to collect ($>$ 250 GB) proof process data [27, 28], which we are currently analysing to identify clusters using AI techniques akin to [16]. We hope with this data to find hard evidence for proof patterns. The AI4FM hypothesis is that *"enough information-extraction can be automated from a mechanical proof that future proofs of examples of the same class can have increased automation"*. The challenge is discussed in several earlier publications including [6, 21, 10]. As such, an important area of future work on proof patterns is the transference of a proof, described using patterns, to help automate another similar proof obligation. We are also interested in extending our catalogue of proof patterns, as well as collecting further examples of the patterns for different formal methods. While we have given a natural language presentation of patterns in this paper, we would like to formalise a pattern language in order to present and automatically generate proofs from the patterns, similar to work in software design patterns [3]. We will analyse the proof process data captured by our tools using AI techniques.

Acknowledgements We are grateful to Cliff Jones for suggesting the Heap problem and for, together with Andrius Velykis, the many fruitful discussions. Other AI4FM members helped us understand important problems in automated reasoning. This work is supported by EPSRC grant EP/H024204/1.

Bibliography

[1] J.-R. Abrial. *The Event-B book*. Cambridge University Press, UK, 2010.

[2] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, later printing edition, August 1977.

[3] F. J. Budinsky, M. A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[4] 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.

[5] 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 *LNCS*, pages 111–120. Springer Berlin, 1988.

[6] Alan Bundy et al. Learning from experts to aid the automation of proof search. In Liam O'Reilly et al., editors, *PreProceedings of the 9th AVoCS'09*, CSR-2-2009, pages 229–232. Swansea University, UK, 2009.

[7] F. Buschmann, K. Henney, and D.C. Schmidt. Past, present, and future trends in software patterns. *Software, IEEE*, 24(4):31–37, 2007.

[8] Frank Buschmann et al. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[9] 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.

[10] Leo Freitas, Cliff B. Jones, Andrius Velykis, and Iain Whiteside. How to say why. Technical Report CS-TR-1398, Newcastle University, www.ai4fm.org/tr, November 2013.

[11] Leo Freitas and Iain Whiteside. Proof patterns for formal methods. Technical Report CS-TR-1399, Newcastle University, Nov 2013.

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

[13] Leo Freitas and Jim Woodcock. A chain datatype in Z. *International Journal of Software and Informatics*, 3(2-3):357–374, 2009.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1998.

[15] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. A graphical language for proof strategies. To appear in LPAR'13. Available at arXiv:1302.6890, 2013.

[16] Jonathan Heras and Ekaterina Komendantskaya. ML4PG in computer algebra verification. In *Conf. on Intelligent Computer Mathematics*, 2013.

[17] Jónathan Heras and Ekaterina Komendantskaya. Statistical proof-patterns in Coq/SSReflect. *CoRR*, abs/1301.6039, 2013.

[18] Mateja Jamnik et al. Automatic learning of proof methods in proof planning. *Logic Journal of the IGPL*, 11(6):647–673, 2003.

[19] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

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

[21] Cliff B. Jones, Leo Freitas, and Andrius Velykis. Ours is to reason why. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods*, volume 8051 of *LNCS*, pages 227–243, 2013.

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

[23] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.

[24] L. Paulson. Isabelle: A Generic Theorem Prover. *LNCS*, 828, 1994.

[25] Mark Saaltink. The Z/EVES system. In Jonathan Bowen et al, editor, *ZUM '97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer Berlin / Heidelberg, 1997.

[26] Nataliia Sleshina. Proof engineering through a proof process. Master's thesis, School of Computing Science, Newcastle University, 2013.

[27] Andrius Velykis. Inferring the proof process. In Christine Choppy et al., editors, *FM2012 Doctoral Symposium*, Paris, France, August 2012.

[28] Andrius Velykis. *Capturing & Inferring the Proof Process (under submission)*. PhD thesis, School of Computing Science, Newcastle University, 2014.

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

[30] Jim Woodcock and Jim Davies. *Using Z*. Prentice Hall International, 1996.