Chapter 7

Supporting Proof in VDM-SL using Isabelle

Sten Agerholm and Jacob Frost

Summary

This chapter describes the construction of a theorem proving component of a prototype integrated CASE and theorem proving tool which combines the benefits of a general-purpose theorem prover called Isabelle with those of a commercial software development environment for VDM-SL—the IFAD VDM-SL Toolbox. The integrated tool supports proof in the notation of the CASE tool by handling "difficult" constructs such as patterns and cases expressions in a novel way using reversible transformations. Hence, it gives the user a consistent view on the modeling, the model analysis and the proof processes as both pragmatic testing and formal proof is supported in one notation. The chapter illustrates the use of the theorem prover on two examples where automation of proof support is a key issue and a challenge due to the three-valued nature of the Logic of Partial Functions (LPF) underlying VDM-SL.

7.1 Introduction

A large part of industry's reluctance towards theorem proving is caused by the "take it or leave it" approach that has been taken when presenting the technology to industry. The focus has traditionally been on fully verified systems, and the theorem prover has been the starting point of discussion. We suggest instead taking a more pragmatic starting point, such as a CASE tool, and step by step "upgrading" this tool with support for proofs. More light-weight use of theorem provers is to "debug" specifications by proving various consistency conditions, such as type checking

conditions in PVS [20] and type checker generated proof obligations in the IFAD VDM-SL Toolbox [5]. More heavy-weight use is, for example, to prove refinements of specifications.

This chapter presents the first steps towards building an industrial-strength proof support tool for VDM-SL using this CASE tool oriented approach. Our starting point is the IFAD VDM-SL Toolbox [17, 8, 11], which is a commercial software development environment that supports a range of development activities, including various static checks, specification level execution and debugging and code generation to C++. We try to combine the benefits of this toolset with the benefits of the generic theorem prover Isabelle[1] [18]. We do not build the theorem prover from scratch since this is a far too time consuming task, and systems like Isabelle are designed to allow fast construction of theorem provers for new logics. The chapter focuses both on the construction of a theorem prover for VDM-SL in Isabelle, called VDM-LPF, and on the integration of this "proof engine" with the IFAD VDM-SL Toolbox in a way that gives the user a consistent view on the specification and the proof process. Our intended use of the combined tool is mainly for proving type consistency proof obligations. Experiments have already shown this to be a powerful approach to debug specifications [2] and to prove safety properties for operations in state-based systems. However, it will also be possible to prove general correctness requirements of specifications.

The first attempt to build proof support for VDM-SL was in the Mural project [13, 6], and these results have been an important starting point for this work, in particular the book [6]. However, our combined tool extends the subset of VDM-SL supported in Mural with (at least) let expressions, cases expressions, patterns, enumerated expressions, quote types and the character type. Difficult constructs like patterns and cases expressions are handled using reversible transformations and special-purpose derived proof rules that mimic the original expressions.

LPF

The "Logic of Partial Functions" (LPF) is a well-established basis for reasoning about VDM-SL specifications [12, 13, 6]. Consequently we have chosen to base the theorem prover component of our system on LPF.

LPF is designed specifically to cope with "undefined values" resulting from partiality of functions. Logics such as first-order classical logic are two-valued in the sense that formulas are either true or false. In contrast, LPF is three-valued, allowing formulas also to be undefined. Because many of the connectives are non-strict, a formula can be defined even though its subformulas are undefined. For example, the formula `e1 or e2` is true whenever one of its subformulas `e1` or `e2` is true even if the other is undefined. To be false both subformulas must be false. In the remaining situations the disjunction is undefined.

[1] A generic theorem prover provides a logical framework in which new logics can be formulated. A new logic is called an instantiation, or an object-logic.

The definition of LPF means that it has many nice properties. For example, both disjunction and conjunction behave symmetrically. In fact, all inference rules valid in LPF are also valid in classical logic. However, the opposite is not true. Most noticeably, the law of the excluded middle `e or not e` does not hold due to the third value representing undefinedness.

Isabelle

Isabelle [18] is a generic theorem proving system which can be instantiated to support reasoning in new so-called *object-logics* by extending its *meta-logic*. The language of the meta-logic is typed lambda-calculus. The syntax of an object-logic is implemented by extending this language with new types and constants. The inference rules of an object-logic are implemented by extending the meta-logic with corresponding meta-axioms. Object-level natural deduction proofs can be carried out as meta-level proofs using different forms of resolution to apply rules.

The Isabelle system contains a range of useful features. For example, it provides unknowns, which are essentially free variables that can be instantiated gradually during a proof by higher-order unification. It also provides syntax annotations, syntax declarations and several translation mechanisms that are useful for handling concrete syntax. In addition, it has a tactic language and generic packages to write powerful proof procedures for object logics with little effort.

Organization of this Chapter

We first give an overview of our approach to build an integrated CASE and theorem proving tool in Section 7.2. The following three sections, Section 7.3 to Section 7.5, describe the Isabelle instantiation: syntax, proof theory and proof support (tactics). Section 7.6 and Section 7.7 concern the integration, respectively the transformation of expressions to fit into the subset supported by Isabelle and the generation of "representations" of specifications as Isabelle theories. Section 7.8 presents future work and Section 7.9 the conclusions. This chapter collects material from the two papers [3, 4].

7.2 Overview of Approach

The overall idea of the integrated system is that a user writes a VDM-SL specification using the IFAD VDM-SL Toolbox to syntax check, type check and possibly validate the specification. When the user wants to prove a property entered by hand or a proof obligation generated by the type checker, he can start the Proof Support Tool (PST), generate axioms from a specification and load these into a VDM-SL instantiation of Isabelle. The PST will then provide a Graphical User Interface (GUI) to Isabelle through which proofs can be conducted and managed in a flexible way. This system architecture is illustrated in Figure 7.1. We call this a two-layered

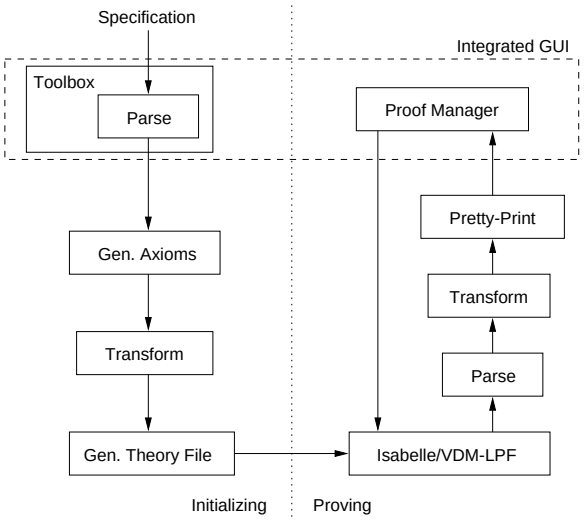

Figure 7.1: Overview of system architecture.

architecture in the sense that the theorem prover is the bottom layer, the "proof engine", and the proof support tool and its graphical user interface are the top layer. In between the layers, transformation, parsing and pretty-printing occur.

7.2.1 Reading of Figure 7.1

The left hand side of Figure 7.1 concerns initializing the proof support tool for a specification. The specification is written and then syntax and type checked using the Toolbox. It must be type checked because axioms are generated on basis of this assumption (see Section 7.7). The resulting abstract syntax tree representation of the specification is communicated to the proof support tool (via a file link). From the syntax tree, axioms are generated as proof rules whose hypotheses are stated using abstract syntax for VDM-SL and type judgements. The expression parts of axioms are then transformed to a subset of VDM-SL. This subset does not contain for instance patterns and cases expressions. Finally, the transformed axioms are printed together with signature information for new constants to an Isabelle theory file.

The theory file generated as above is read into an instantiation of Isabelle, called Isabelle/VDM-LPF, which contains a proof theory for the VDM-SL subset. The right hand side of Figure 7.1 illustrates how proofs of theorems and proof obligations are conducted by sending commands to Isabelle via the PST graphical user interface. Such commands could tell Isabelle to start a new proof or to apply some tactic in an on-going backwards proof. Proof states in backwards proofs are represented as proof rules in Isabelle and the Toolbox parser has been extended to read Isabelle's notation for these. After parsing proof states to abstract syntax, a transformation function is applied, which reintroduces for instance patterns and cases expressions.

Finally, a VDM-SL ASCII pretty-printer produces concrete syntax which can be displayed to the user. Both the generation of axioms, the transformation and the pretty-printing are specified in VDM-SL itself. The underlying "programs" work by executing specifications using the Toolbox.

7.3 Syntax

Isabelle is a generic theorem prover which provides distinguishing features for embedding new logics (syntax and proof rules) and reasoning in these logics. Isabelle has a meta-logic, an intuitionistic higher order logic based on typed lambda-calculus, in which constructs of new logics become constants, primitive proof rules become (meta-) axioms, and derived rules become (meta-) theorems. Moreover, it provides a few, but powerful tactics for applying these rules, based on different kinds of resolution and generic packages for writing proof search procedures for new logics (see Section 7.5).

When embedding a language such as VDM-SL in Isabelle, two sometimes conflicting criteria must be met. Firstly, the language should allow smooth reasoning using features of Isabelle. Secondly, it should provide a user-friendly and familiar notation. These two goals are achieved by implementing an abstract and a concrete syntax and by relating these using automatic translations.

Syntax issues are described in more detail in the Appendix. Here, we just illustrate on some examples that, with our definitions, Isabelle is able to print and parse exactly the VDM-SL concrete syntax such as the following[2]:

```
[3,1,2]
{x + 1 | x in set {1,2,3} & x >= 2}
forall x:nat, y:nat & x + y = y + x
let x = 5, y = x + 6, z = 7 in y + z
if n = 0 then 1 elseif n = 1 then 2 elseif n < 10 then 9 else n
mk_(x,5,z).#2 + mk_(1,2,3,4).#3 = 8
```

As illustrated in the last line, Isabelle's syntax handling features support something as exotic as arbitrary-length tuples and generalized projections.

7.4 Proof System of VDM-LPF

The proof system for VDM-SL axiomatized in Isabelle has been developed with minor modifications from that presented in the book [6], where LPF is used as the basis. In addition to primitive and derived rules for propositional and predicate LPF, the book contains a large number of rules for datatypes such as natural numbers, sets, sequences, maps, etc. As in many other formulations of LPF, these rules are

[2] Throughout this chapter we use ASCII syntax for VDM-SL in order to highlight the similarity between the syntax used by the IFAD Toolbox and the Isabelle instantiation.

formulated as natural deduction rules. This fits well with the choice of Isabelle which supports natural deduction style proof particularly well.

The following two subsections describe part of the VDM-SL proof system and its implementation in Isabelle. The axiomatization of, for example, datatypes is not discussed since we do not contribute to the presentation in [6].

7.4.1 Proof Rules

The proof system of LPF contains most of the standard rules of propositional and predicate logic. However, it is a three-valued logic so it does not provide the law of excluded middle, and some rules have additional typing assumptions, for example, in order to ensure that equality is defined.

Most of the rules of the Isabelle instantiation are taken from [6]. The adaptation consists mostly of changing the syntax. For example, consider the rules:

$$\frac{}{\mathsf{true}} \qquad \frac{P;\ Q}{P \wedge Q} \qquad \frac{P \vee Q;\ P \vdash R;\ Q \vdash R}{R}$$

$$\frac{x : A \vdash_x \neg P(x)}{\neg(\exists\, x \cdot : A \cdot P(x))} \qquad \frac{b : A;\ a = b;\ P(b)}{P(a)}$$

These rules are translated into the following Isabelle axioms:

```
true_intr       "true"
and_intr        "[| P; Q |] ==> P and Q"
or_elim         "[| P or Q; P ==> R; Q ==> R |] ==> R"
not_exists_intr "(!!x.x:A ==> not P(x)) ==> not (exists x:A & P(x))"
eq_subs_left    "[| a=b; P(b); b:A |] ==> P(a)"
```

In these axioms, `==>` represents meta-implication while `[|` and `|]` enclose a list of assumptions separated by `;`. Note that variable subscripts for locally bound variables in sequents are handled using meta-quantification `!!`. Apart from using the concrete ASCII syntax of VDM-LPF (see Appendix), these examples illustrate how the horizontal line and turnstile in the original rules are represented as meta-implication.

In some cases, the order of assumptions is changed to make the rules work better with Isabelle's resolution tactics, which often use unification on the first assumption to instantiate variables. In particular, type assumptions are often moved from the front of the assumption list to the back, since these do not typically contain any important information for restricting the application of rules. An example of this is the last substitution rule above.

7.4.2 Combining Natural Deduction and Sequent Style Proof

In order to formalize proof rules as meta-level axioms in Isabelle it is necessary to define a judgement relating object-level formulas to meta-level formulas in Isabelle's

higher-order logic. In the LPF variant used here, there is no distinction between expressions and formulas. Hence, expressions (and formulas) are represented as lambda-calculus terms of type `ex`. In Isabelle meta-level formulas are terms of type `prop`. Consequently, the standard judgement relating object-level expressions to meta-level formulas is the following lifting constant:

```
TRUE' :: ex => prop   ("(_)" 5)
```

The concrete syntax associated with this constant in brackets specifies that `TRUE'` will not occur explicitly in (meta-) axioms representing proof rules (5 is a precedence). Hence, `TRUE'` was invisible in the proof rules presented above.

The standard judgement `TRUE'` is sufficient for implementing a natural deduction system for LPF. However, in order to automate proving in VDM-LPF, it is advantageous to be able to conduct (or simulate) sequent calculus style backward proof. In classical logic this can be done by representing multiple conclusions as negated assumptions. This issue is a consequence of the law of excluded middle. However, LPF is a three-valued logic and does not satisfy this law. Instead, we have therefore declared an additional judgement for this purpose:

```
FALSE' :: ex => prop   ("FALSE _" 5)
```

In this case the concrete syntax requires an object-level formula to be preceded by `FALSE` (specified in brackets). The idea is that this judgement acts as a kind of non-strict negation with respect to the third value of LPF. This form of negation, which can only occur at the outermost level, allows multiple conclusions to be represented as a "negated" assumptions. This is discussed further in Section 7.5.

There are two new rules for the `FALSE` judgement:

```
FALSE_dup   "(FALSE P ==> P) ==> P"
FALSE_contr "[| FALSE P; P |] ==> Q"
```

The first rule allows a conclusion to be duplicated as a negated assumption using `FALSE`, while the second rule is a contradiction-like rule. Together these rules imply that `FALSE` behaves as desired.

The inclusion of the additional judgement `FALSE` to represent multiple conclusions has interesting consequences for the proof system. For example, the primitive rule

```
(!! y.y:A ==> def P(y)) ==> def exists x:A & P(x)
```

becomes derivable, and it is no longer necessary or useful. This is fortunate since the rule appears to be hard to apply properly, especially in automatic backwards proof. An automatic proof of the rule is given in Section 7.5.2.

Soundness

We are confident, but have not formally proved, that the modifications of the proof system discussed above are sound. We base this confidence on Cheng's thesis [7], who

has proved essentially this result for VDM-SL without datatypes. Cheng formulates both a sequent calculus and a natural deduction proof system for predicate LPF with equality and non-strict connectives[3]. Among other things, he proves that these proof systems are equivalent in terms of what can be derived. However, he does not consider datatypes like those of VDM-LPF. Hence, we should eventually extend Cheng's work to ensure that the VDM-SL datatype extensions of LPF do not violate soundness. We have done some preliminary work on this.

7.5 Proof Tactics

Isabelle's built-in tactics can be used immediately with VDM-LPF for fine-grained proof. However, it is advantageous to build special-purpose tactics exploiting the `FALSE` judgement as well as tactics for doing automatic proof search. This section describes such basic tactics as well as proof search tactics.

7.5.1 Basic Tactics

In many cases Isabelle's standard resolution tactics are suitable for applying the rules of VDM-LPF directly. For example, consider a proof state with the following subgoal:

```
1.  Q and P ==> P or Q
```

The conclusion of this goal can be broken down using the standard resolution tactic `resolve_tac` with the or-introduction rule `or_intr_left`, which will apply this rule to the conclusion in a backward fashion. This can be written using Isabelle's useful shorthand `br`, where `b` stands for `by`:

```
- br or_intr_left 1;  (* same as: by(resolve_tac [or_intr_left] 1) *)
1.  Q and P ==> P
```

In a similar fashion the assumption of the subgoal can be broken down, this time using `eresolve_tac` to apply the and-elimination rule `and_elim` in a forward fashion on the assumption of the subgoal:

```
- be and_elim 1;  (* same as: by (eresolve_tac [and_elim] 1) *)
1.  [| Q; P |] ==> P
```

Finally, the remaining subgoal can be solved using `assume_tac` which simulates proof by assumption in natural deduction:

```
- ba 1;  (* same as: by (assume_tac 1) *)
No subgoals!
```

[3]The connectives, in particular definedness (for "true or false"), in our variant of LPF are all strict.

In addition to the above tactics, Isabelle has a few other basic resolution tactics for applying natural deduction rules (see [18]).

There are a few situations where the above tactics are not suitable for applying the rules of VDM-LPF. For example, consider the following subgoal where a multiple conclusion is represented as an assumption using the `FALSE` judgement:

```
1.  [| P and Q; FALSE P or Q |] ==> R
```

In this case `resolve_tac` cannot be used directly to apply `or_intr` to (part of) the conclusion `P or Q`, since it is represented as an assumption. Instead we have developed variants of the standard resolution tactics, which can be used in such situations. The names of these are obtained by just adding an additional prime on standard names, as in `resolve_tac'`. These tactics use the proof rules for the `FALSE` judgement to allow a rule to be applied to conclusions represented in assumptions using `FALSE`. In addition to these, a VDM-LPF variant of `assume_tac`, called `assume_tac'`, allows one step proofs of subgoals like:

```
[| P |] ==> P
[| P; FALSE P |] ==> R
[| P; not P |] ==> R
```

The first case is just ordinary proof by assumption, the next covers the situation where a conclusion appears in the assumptions due to `FALSE`, while the last case deals with the situation where a proof branch is ended by an application of the primitive LPF contradiction rule `[| P; not P |] ==> Q`.

7.5.2 Proof Search Tactics

Isabelle also provides a generic classical reasoning package for automating larger parts of proofs than supported by the tactics above. However, VDM-LPF is not classical and therefore it seems hard to use this package. Classical laws are used for simulating sequents using natural deduction. Instead we have implemented a new package designed specifically for VDM-LPF. This package combines ideas from Cheng's thesis on LPF [7] with ideas and code from the classical reasoning package. As in Isabelle, the aim of this is to provide a practical tool and less emphasis is put on completeness issues.

The proof search tactics in the packages are based on the same basic idea: do sequent calculus style backward proofs using suitable rules to break assumptions and conclusions of subgoals gradually down, until the conclusion is provable from an assumption. In other words, natural deduction introduction rules are applied as right sequent style rules, while elimination rules are applied as left rules. Our package handles multiple conclusions using `FALSE` and by working as if using the primed versions of the tactics. Rules for both `FALSE` and strict negation `not` are required by the package. In contrast, the Isabelle classical reasoning package does not make such a distinction since it uses just classical negation to represent multiple conclusions (this is not possible in non-classical logics).

Rules applied by the tactics are organized in rule sets and are supplied directly as arguments of the tactics. As in Isabelle's classical reasoner, a rule set is constructed by dividing rules into groups of introduction rules, elimination rules, etc. For each of these groups, the rules are further divided into safe and hazardous rules. Roughly speaking, the idea is that a safe rule can always be attempted blindly, while a hazardous rule might, for example, sidetrack the search (thus requiring backtracking) or cause the search to loop. The search tactics generally try safe rules before the hazardous ones.

In most cases, the grouping is fairly straightforward, for example, consider the following rules for disjunction:

```
or_intr_left  "P ==> P or Q"
or_intr_right "Q ==> P or Q"
or_intr       "[| FALSE P ==> Q |] ==> P or Q";
or_elim       "[| P or Q; P==>R; Q==>R |] ==> R"
not_or_intr   "[| not P; not Q |] ==> not (P or Q)"
not_or_elim   "[| not (P or Q); [| not P; not Q |] ==> R |] ==> R"
```

The first two rules are hazardous introduction rules, since they force a choice between the two disjuncts. In contrast the third is a safe introduction rule using `FALSE` to represent the two possible conclusions. The fourth rule is just the standard elimination rule for disjunction. In addition to these, rules explaining how negation behaves when combined with disjunction are needed. This is the purpose of the last two rules. The reason why these are needed is that it is not possible to give general rules for negation. These conjunction-like introduction and elimination rules are both safe. The rule set `prop_lpfs` for propositional VDM-LPF contains all the above rules, except the hazardous introduction rules which are replaced by the single safe one.

In order to illustrate how the search strategy described above can be used to find proofs, a sketch of a small proof following this strategy is shown below:

```
1.  not (Q and P) ==> not P or not Q

- br or_intr 1;
1. [|  not (Q and P); FALSE  not P |] ==>  not Q

- be not_and_elim 1;
1. [| FALSE  not P;  not Q |] ==>  not Q
2. [| FALSE  not P;  not P |] ==>  not Q

- ba' 1; ba' 1;  (* same as: by (assume_tac' 1) *)
No subgoals!
```

The tactic `lpf_fast_tac` combines the strategy above and depth first search with backtracking at suitable points (e.g. if more than one unsafe rule is applicable). This is probably the most used tactic in the package. For example, when invoked as `lpf_fast_tac prop_lpfs 1`, this tactic proves the above theorem in one step.

Other similar tactics support safe steps only to be carried out (`lpf_safe_tac`), support the restriction of the depth of proofs (`lpf_depth_tac`), etc. So far these tactics have been used to prove (in one step) essentially all of the 120 derived propositional and predicate logic rules mentioned in [6].

Before we consider a "real" case study, we end this subsection by considering a simple example where the assumption of the rule is itself a generalized rule:

```
val [asm] =
goal Pred.thy "(!!y.y:A ==> def P(y)) ==> def exists x:A & P(x)";
by (lpf_fast_tac (exists_lpfs addDs [asm]) 1);
qed "def_exists_inh";
```

This situation is handled by adding the assumption to `exists_lpfs`, which contains a number of rules about the exists quantifier and is one of the intermediate rule sets used to build the theory of predicate VDM-LPF. Hence, the proof search tactics provide a universal strategy which also works for most proofs about quantifiers as long as a sufficient quantifier rule set is supplied as an argument to guide the search. Moreover, this example shows that `lpf_fast_tac` can be used to prove a quantifier rule which is primitive in LPF (see Section 7.4.2).

7.5.3 Gateway Example

This section illustrates how the VDM-LPF instantiation works on an example VDM-SL specification of a trusted gateway (provided by John Fitzgerald). Though the example is small, it is inspired by an industrial case study on formal methods [10, 14]. Strictly speaking, we do not support reasoning about the specification itself, but about an axiomatization of the specification which is automatically generated as described in Section 7.7. The details of the axiomatization are not important for this section.

In the following we first present excerpts from a specification of the trusted gateway and then illustrate the use of our proof tactics to prove invariant and safety properties. We use ASCII syntax for both the Toolbox's and Isabelle's VDM-SL since it is an important point that they are the same.

A Trusted Gateway

A trusted gateway connects an input port to two output ports, which are a high- and a low-security port (see Figure 7.2). The purpose of the gateway is to prevent accidental disclosure of classified or sensitive information on the low-security port. The trusted gateway reads messages from the input port into an internal block where it analyzes them according to two categories of high- and low-classification strings. It must send messages containing only low-classification strings to the low-security port and all other messages to the high-security port.

In VDM-SL we can model a trusted gateway as the following record type:

Figure 7.2: A Trusted Gateway

```
Gateway:: input: Port
          highOutput: Port
          lowOutput: Port
          high: Category
          low: Category
          block: Message
inv g == (forall msg in set elems g.highOutput &
            classification(msg,g.high,g.low) = <HIGH>) and
         (forall msg in set elems g.lowOutput &
            classification(msg,g.high,g.low) = <LOW>) and
         g.high inter g.low = {}
```

This type has an invariant which specifies correct behaviour of gateways, i.e. all messages on the high-security output port must be classified as high, all messages on the low-security output port must be classified as low, and the two categories of high- and low-classification should not have any strings in common. A message is modeled as a sequence of strings, a port as a sequence of messages, and a category as a set of strings.

The different operations on the gateway are modeled as functions acting on elements of type `Gateway`. These include a function for loading an element into the internal block, a function for analyzing the contents of the block and a function for emptying the block. For illustration, the specification of the analyze function is shown:

```
Analyze: Gateway -> Gateway
Analyze(g) ==
  if classification(g.block,g.high,g.low) = <HIGH>
  then mk_Gateway(g.input,
                  g.highOutput ^ [g.block],
                  g.lowOutput,
                  g.high,g.low,g.block)
  else mk_Gateway(g.input,
                  g.highOutput,
                  g.lowOutput ^ [g.block],
                  g.high,g.low,g.block);
```

This function copies messages from the internal block to the appropriate output port, depending on the result returned by the classification function. The definition of the classification function is central:

```
classification: String * Category * Category -> (<HIGH> | <LOW>)
classification(s,high,low) ==
```

```
if contains_category(s,high)
then <HIGH>
elseif contains_category(s,low)
then <LOW>
else <HIGH>
```

Note that the order of the if-conditions is important, and messages that contain neither high nor low classified strings are classified as high for security reasons.

Invariant Properties

In order to be well-formed, functions such as `Analyze` must respect the invariant of their result type. For `Analyze`, this means that its body must yield a gateway assuming that its argument is a gateway. This is stated as follows in Isabelle:

```
g : Gateway ==>
if classification@(g.block, g.high, g.low) = <HIGH> then
  mk_Gateway(g.input, g.highOutput ^ [g.block], g.lowOutput,
    g.high, g.low, g.block)
else
  mk_Gateway(g.input, g.highOutput, g.lowOutput ^ [g.block],
    g.high, g.low, g.block) :
Gateway
```

Note that essentially the same ASCII syntax is supported by Isabelle here and by the VDM-SL Toolbox in the previous section, but object-level application in VDM-LPF is written using `@` instead of the usual juxtaposition (discussed further in Section 7.7.1). Also note that postfix field selection is supported in Isabelle by declarations such as

```
block' :: ex => ex  ("_.block" ...)
high' :: ex => ex  ("_.high" ...)
```

produced by the axiomatization. The above statement, which is generated automatically using the proof obligation generator described in [5], has been proved in VDM-LPF. However, due to lack of space it is not possible to show the entire interactive proof here. Instead some central steps in the proof are discussed.

The first obvious step is to use case analysis on the conditional expression. Due to the third value in VDM-LPF, we get three subgoals, the two obvious ones and another which says that the test condition must be true or false (it could potentially be undefined in some cases). Here we concentrate on the subgoal corresponding to the first branch of the conditional:

```
[| g : Gateway; classification@(g.block,g.high,g.low) = <HIGH> |] ==>
mk_Gateway
 (g.input, g.highOutput^[g.block], g.lowOutput, g.high,
  g.low, g.block) :
Gateway
```

To prove this, we first unfold the type definition of `Gateway` in the assumptions and the conclusion, then we prove some trivial type judgement subgoals automatically, and what we obtain is the subgoal:

```
[| classification@(g.block,g.high,g.low) = <HIGH>; g.input : Port;
   g.highOutput : Port; g.lowOutput : Port; g.high : Category;
   g.low : Category; g.block : Message;
   (forall m in set elems g.highOutput &
      classification@(m,g.high,g.low) = <HIGH>) and
   (forall m in set elems g.lowOutput &
      classification@(m,g.high,g.low) = <LOW>) and
   g.high inter g.low = {} |] ==>
(forall m in set elems (g.highOutput^[g.block]) &
   classification@(m,g.high,g.low) = <HIGH>) and
(forall m in set elems g.lowOutput &
   classification@(m,g.high,g.low) = <LOW>) and
g.high inter g.low = {}
```

The hardest part is clearly to prove the first conjunct of the conclusion, the other two follow trivially from the assumptions. One tactic `lpf_fast_tac` proves the above subgoal:

```
by (lpf_fast_tac
    (prop_lpfs addIs [foralls_lemma] addEs [Port_elim]) 1);
```

Its rule set argument for guiding the proof search contains proof rules for propositional reasoning, a suitable lemma for the difficult conjunct and a single rule for type checking ports. The lemma states how to break the quantification according to the sequence concatenation, and it is proved from various properties about sets, sequences and of course the set-bounded universal quantifier.

Safety Property

The safety property for a trusted gateway states that all messages on the low security output port are given a low security classification:

```
g : Gateway ==>
forall m in set elems g.lowOutput &
  classification@(m, g.high, g.low) = <LOW>
```

This has a single tactic proof:

```
by (lpf_fast_tac
    (prop_lpfs addEs [Gateway_elim] addDs [inv_Gateway_dest]) 1);
```

The rule set argument supports propositional reasoning, unfolding of the gateway type assumption and unfolding of the resulting gateway invariant. Inspecting the property more closely, it is not surprising that the proof turned out to be that simple, since the safety property is essentially a part of the invariant. Thus the real

challenge is to prove that the invariant on `Gateway` is preserved.

7.6 Transformations

We have now completed the description of the VDM-SL proof engine based on Isabelle. The following two sections concern aspects related to the integration of this proof tool with the IFAD VDM-SL Toolbox.

As already noted, the proof tool supports a subset of VDM-SL, since VDM-SL was designed for writing large specifications in industry with constructs that are not designed for embedding in a theorem prover. This section describes how we treat some of the complex constructs by transforming expressions down to expanded expressions in the subset that has been formalized in Isabelle.

Throughout this section we use a record type `R` of the following form:

```
R :: a:T1
     b:T2
     c:T3
```

7.6.1 Pattern Matching

We represent pattern matching by transforming patterns to combinations of if and let expressions where a boolean condition for matching is expressed using an existential quantification. The user of the system does not see these expanded forms since the expansion is reversed before showing output from Isabelle to the user (after this has been parsed by the Toolbox parser).

In this section we illustrate how to treat patterns in let, quantified and comprehension expressions. In the presentation, we just use expression templates, but the transformation does work for real specifications, see Section 7.7.

Let Expressions

The VDM-LPF instantiation of Isabelle supports only simple let expressions of the form

```
let x = e1, y = e2[x], z = e3[x,y] in expr
```

where `x`, `y` and `z` are variables. Note that the first variable of the let expression may be used in the second expression, etc.

However, a VDM-SL let expression may contain general patterns in addition to variables. Consider a let expression of the form

```
let mk_R(x,-,z) = e1,
    mk_((e),y) = e2
in body
```

Here, `x`, `z` and `y` are pattern variables, `-` is the don't care pattern and `(e)` is a value pattern, i.e. the expression `e` is evaluated. The expression `e1` must be a record of type `R` for the first pattern to match, and for the second pattern to match, `e2` must be a tuple with two components where the first component is equal to the value of `e`. If one of the patterns does not match, then the expression is undefined.

The above expression is transformed to an equivalent expanded expression containing if expressions, existential quantifiers and simple let expressions:

```
let new = e1 in
if (exists x:T1, z:T3, dc:T2 & mk_R(x,dc,z) = new) then
  let x = new.a, z = new.c in
  let new2 = e2 in
  if (exists y:Ty & mk_(e,y) = new2) then
    let y = new2.#2 in body
  else undefined
else undefined
```

This expression lies within the VDM subset formalized in Isabelle; `undefined` is a constant symbol defined with no rules and represents a divergence from [6]. The variables `new`, `new2` and `dc` must be new identifiers generated by the transformation (see Section 7.7). The `&` ends a binding list and corresponds to "such that". The selector `#2` takes the second component of a tuple. The type information on, for example, `y` in the second exists is available in the Toolbox abstract syntax tree after type checking. Note that we illustrate the transformation on concrete syntax due to readability, but it is really performed on the abstract syntax tree.

Quantified Expressions

Patterns in quantified expressions are represented using an existential quantifier and an implication `=>`. Consider for example

```
forall mk_(mk_R(x,-,z),(e)) in set s & expr[x,z]
```

which is transformed to:

```
forall new in set s &
  (exists x:T1, z:T3, dc:T2 & mk_(mk_R(x,dc,z),e) = new) =>
  let x = new.#1.a, z = new.#1.c in expr[x,z]
```

For existential quantification we must replace implication by conjunction.

Comprehensions

Set, map, and sequence comprehensions can all be treated in the same way. For example the set comprehension

```
{expr[x,z] | mk_(mk_R(x,-,z),(e)) in set s & p[x,z]}
```

is transformed to:

```
{let x = new.#1.a, z = new.#1.c in expr[x,z] | new in set s &
 exists
   x:T1, z:T3, dc:T2 & mk_(mk_R(x,dc,z),e) = new and p[x,z]}
```

This is also illustrated in the example in Section 7.7.

7.6.2 Cases Expressions

Cases expressions are not supported in the instantiation of Isabelle. VDM-SL cases expressions are transformed to certain combinations of if and simple let expressions, in a way somewhat similar to let expressions with patterns. However, to the user these combinations appear to be cases expressions since the transformation to expanded expressions is reversed before expressions are displayed. Moreover, there are proof rules which make the if-let combinations appear to be cases expressions.

Consider the following example which uses the same patterns that were used above:

```
cases expr:
  mk_R(x,-,z) -> e1,
  mk_((e),y) -> e2,
  others -> e3
end
```

This is transformed to the following expression:

```
let new = expr in
if (exists x:T1, z:T3, dc:T2 & mk_R(x,dc,z) = new) then
  let x = new.a, z = new.c in e1
elseif (exists y:Ty & mk_(e,y) = new) then
  let y = new.#2 in e2
else e3
```

One difference between the expanded forms of let and cases expressions is the use of undefined `else` branches in if expressions used to represent let expressions.

A number of proof rules has been derived to make the expanded forms mimic cases expressions, i.e. these rules can be applied on let-if combinations in such a way that it appears to the user that these are real cases expression (due to reverse transformation):

```
cases_match
  "[| P(e); e:A; e1(e):B |] ==>
   (let x = e in if P(x) then e1(x) else e2(x)) = e1(e)"
cases_not_match
  "[| not P(e); e:A; e2(e):B |] ==>
   (let x = e in if P(x) then e1(x) else e2(x)) =
   (let x = e in e2(x))"
cases_form_sqt
```

```
"[| def P(e); e:A;
    [| P(e) |] ==>  e1(e):B;
    [| not P(e) |] ==> let x = e in e2(x):B |] ==>
 (let x = e in if P(x) then e1(x) else e2(x)) : B"
```

Let us explain why these proof rules mimic cases expressions from the viewpoint of a user. For simplicity we consider a slightly different example than above. Suppose we are doing a proof where we would like to reduce a cases expression of the form:

```
cases ex:
  mk_(n,m,m) -> n+m,
  mk_(n,-,n) -> n,
  others     -> 0
end
```

Further, suppose that we expect the first pattern to not match. We would then use the proof rule `cases_not_match` to reduce the cases expression. In conducting a proof this rule would be instantiated to:

```
"[| not exists n:nat, m:nat & mk_(n,m,m) = ex; ex:A;
    if (exists n:nat, dc:nat & mk_(n,dc,n) = ex) then n
    else 0:B |] ==>
 (let new = ex in
  if (exists n:nat, m:nat & mk_(n,m,m) = new) then
    let n = new.#1,
        m = new.#2 in n+m
  elseif (exists n:nat, dc:nat & mk_(n,dc,n) = new) then
    let n = new.#1 in n
  else 0) =
 (let new = ex in
  if (exists n:nat, dc:nat & mk_(n,dc,n) = new) then n else 0)"
```

Isabelle would typically do this instantiation and the necessary β-reductions automatically, a user just tells it to apply the rule (by providing the name of the rule). It is important to realize that the left-hand side of the equality is the expanded form of the original cases expression and that the right-hand side of the equality is still a cases expression, namely:

```
cases ex:
  mk_(n,-,n) -> n,
  others     -> 0
end
```

Hence, the first cases expression above reduces to a cases expression where the first pattern has been thrown away, if this pattern does not match. Note that the if-then-else structure of the derived cases rules works because

```
if b1 then e1
elseif b2 then e2
```

```
else e3
```

is just syntactic sugar for

```
if b1 then e1
else if b2 then e2
     else e3
```

7.7 Generating Axioms: An Example

In order to support reasoning about specifications, we automatically generate axioms stated as proof rules which formalize the meaning of the definitions in a specification. These axioms are stated in Toolbox abstract syntax extended with a construction for proof rules, type judgements and subtypes. Once the axioms have been transformed to the Isabelle subset, they can be read into Isabelle and then used in proofs. We shall not go into the details here of the specification of the axiom generator, which was done in VDM-SL and developed using the Toolbox itself, as it is straightforward and based on [6]. Note that in addition to axioms for specifications, Isabelle needs the signatures of new constants. These are also straightforward to generate, again the specification of this was done in VDM-SL using the Toolbox.

In this section we illustrate the working of the axiom (and signature) generator on a small example, which is adapted from an example of a forthcoming book on VDM-SL [9] and inspired by a real industrial system. The example concerns an alarm paging system for a chemical plant. A safety requirement of the system is that for any period of time and any possible alarm code (and location) there must be an expert with the required qualifications to deal with the alarm on duty according to a certain plan.

The specification below is stated in the expression (or functional) subset of VDM-SL, but we could as well have used a state to model the plant and defined some of the functions as implicit operations on the state (using preconditions and postconditions). The approach to state definitions and implicit operations is (also) borrowed from the Mural project. A state definition is treated in essentially the same way as a record and implicit operations are treated as functions on the state which take the state as an extra argument and result.

7.7.1 Type Definitions

We shall model the chemical plant as a record type whose invariant specifies the safety requirement:

```
Plant :: plan   : Plan
         alarms : set of Alarm
inv mk_Plant(plan,alarms) ==
    forall per in set dom plan, alarm in set alarms &
```

```
        QualificationsOK(alarm,plan(per));
```

The record has two fields, containing a plan and a set of alarms respectively. The function `QualificationsOK` defines the safety requirement and will be specified later.

A plan is simply a map from periods to sets of experts:

```
Plan = map Period to set of Expert
inv plan == forall exs in set rng plan & exs <> {};

Period = token;
```

The invariant says that there should always be at least one expert associated with any period. The data type of periods is left unspecified (using the `token` type which just denotes an infinite set with equality).

An expert has an ID and a set of qualifications:

```
Expert :: expertid : ExpertId
          quali    : set of Qualification
inv mk_Expert(-,q) == q <> {};

ExpertId = token;

Qualification = <Elec> | <Mech> | <Chem> | <Chief>;
```

Qualifications are modeled as an enumeration type. An expert must have at least one qualification.

The datatype of alarms is modeled as a record type with two fields, one for the alarm codes and one for the location of the alarms:

```
Alarm :: code : AlarmCode
         loc  : Location;

AlarmCode = <A> | <B> | <C>;

Location = <P1> | <P2> | <P3> | <P4> | <P5> | <P6> | <P7>
```

A function is clearly needed to specify which qualifications are required for the different alarms, this is done below.

We shall now consider the signature of new constants and axioms generated for these type definitions. We cannot include all axioms in this chapter. Axioms are generated by executing a shell script `genax` with the file name of our specification `alarm.vdm` as an argument. The result is a file `alarm.thy` which contains the Isabelle theory file for the specification. This is ready to be loaded into the Isabelle/VDM-LPF instantiation, which is started by executing the binary `vdmlpf` in a shell.

The theory file starts with the signature of new constants. For a record type, like `Plant` used for illustration in the following, a record constructor function is

introduced:

```
mk_Plant :: [ex,ex] => ex
```

This means that `mk_Plant` is a function in the Isabelle meta-logic, which takes two expressions as arguments, corresponding to the two fields, and yields an expression as a result, corresponding to a record with these two field values. The field selectors of the record type are defined in both an abstract and a concrete syntax version which allows the concrete syntax to support VDM-SL field selection using a postfix notation:

```
plan' :: ex => ex   ("_.plan" ...)
alarms' :: ex => ex   ("_.alarms" ...)
```

The primes are used on constants of the abstract syntax by convention. The parenthesis on the right specifies a priority grammar for the postfix notation, we shall not go into details here. For a record type with an invariant, the signature of the invariant function must also be included:

```
inv_Plant :: ex
```

In contrast to the constructor and selector functions this is represented as an object-logic function. A special object-logic application operator is defined in VDM-LPF and written using an `@`. The invariant function could be represented as a meta-logic function but there are technical reasons for not doing this. VDM-SL map application and sequence indexing have to be represented in the object-logic, since maps and sequences are object logic values. And the application of invariant functions, as well as many other functions, is not distinguished from these first forms of applications in the abstract syntax tree of the IFAD Toolbox. Hence, different translations of such equivalently represented constructs would be difficult.

After the signature declarations the generated axioms are stated as proof rules. The definition axiom for the invariant function is stated as follows:

```
inv_Plant_defn
  "inv_Plant@(mk_Plant(plan, alarms)) ==
   forall per in set dom plan, alarm in set alarms &
     QualificationsOK@(alarm, plan@(per))"
```

This is stated as a simple meta-equality rewrite rule. A number of formation and definition axioms are introduced for record types. For the record constructor function, the following two axioms are needed:

```
mk_Plant_form
  "[| gax41 : Plan; gax42 : set of Alarm;
      inv_Plant@(mk_Plant(gax41, gax42)) |] ==>
   mk_Plant(gax41, gax42) : Plant"

mk_Plant_defn
  "gax5 : Plant ==> mk_Plant(gax5.plan, gax5.alarms) = gax5"
```

The gax variables may look strange, they are automatically indexed by the axiom generator. Each field selector of a record type yields two similar axioms:

```
plan_Plant_form
  "gax5 : Plant ==> gax5.plan : Plan"

plan_Plant_defn
  "[| gax41 : Plan; gax42 : set of Alarm |] ==>
   mk_Plant(gax41, gax42).plan = gax41"
```

Many typing hypotheses appear in order to ensure definedness of object equality =.

Basic type definitions like the definition of Plan and AlarmCode yield fewer axioms. Hence, for Plan just two definition axioms are generated:

```
inv_Plan_defn
  "inv_Plan@(plan) == forall exs in set rng plan & exs <> {}"

Plan_defn
  "Plan ==
   << gax1 : map Period to set of Expert & inv_Plan@(gax1) >>"
```

The latter defines Plan as a subtype of a map type, restricted using the invariant of Plan. The AlarmCode definition just yields one axiom

```
AlarmCode_defn
  "AlarmCode == <A> | <B> | <C>"
```

but as a side effect axioms are also generated for each of the quote types, for example:

```
A_axiom
  "<A> : <A>"

A_singleton
  "gax47 : <A> ==> gax47 = <A>"

A_B_disjoint
  "<A> <> <B>"
```

Disjointness axioms are needed for all pairs of quotes. In VDM-SL, the notation <A> is used for both a quote type and its one element. This can be supported in Isabelle since the corresponding constants of the meta-logic have different meta-types. Ambiguities are resolved by the Isabelle type checker.

7.7.2 Function Definitions

We shall now continue the example from above by specifying a few functions. As already noted we need a function to map alarm codes and locations to qualifications required of experts. We also need a function to specify the safety requirement, and

a function to page a set of experts to handle a specific alarm.

However, consider first a simpler function that tests whether a given expert is on duty in a certain period:

```
OnDuty: Expert * Period * Plant -> bool
OnDuty(ex,per,plant) ==
  ex in set plant.plan(per)
pre per in set dom plant.plan;
```

This function has as precondition that the period is valid, i.e. it is covered by the plan. This ensures that the map application in the body of the function is defined.

All functions of specifications are represented as object-logic functions, just like invariants. Hence, the signature of `OnDuty` is simply:

```
OnDuty :: ex
```

The signature of the precondition function `pre_OnDuty` is the same.

A function definition like this, with a precondition and in the explicit style, results in two axioms, one defining the precondition and one defining the function:

```
pre_OnDuty_defn
  "pre_OnDuty@(ex, per, plant) == per in set dom plant.plan"

OnDuty_defn
  "[| ex : Expert; per : Period; plant : Plant;
      pre_OnDuty@(ex, per, plant) |] ==>
   OnDuty@(ex, per, plant) = ex in set plant.plan@(per)"
```

Strictly speaking the latter should include a type judgement in the hypotheses saying that the body of the function is well-typed, in order to ensure that the LPF equality is defined. However, as mentioned in Section 7.2, we assume that the specifications have been checked using the Toolbox type checker (extended with proof obligations), and for convenience we shall therefore omit this condition. Note that we could include the condition as a hypothesis, prove the hypothesis once and for all, and then derive the above axiom. This integration could be taken further in a very tightly integrated system where the Toolbox type checker could be integrated even more with the proof process to "prove" type judgements arising in proofs.

The following function interprets alarms:

```
AlarmQualifications: Alarm -> set of Qualification
AlarmQualifications(alarm) ==
  cases alarm:
    mk_Alarm(<A>,-)   -> {<Mech>},
    mk_Alarm(<B>,loc) ->
        if loc in set {<P1>,<P3>,<P7>}
        then {<Chief>,<Elec>,<Chem>}
        else {<Elec>},
```

```
    others -> {<Chief>}
  end;
```

Alarms with code <A> require a mechanic, regardless of the location (specified using a "don't care" pattern). Alarms with code <B> at the locations <P1>, <P3> and <P7> are more serious, they require three qualifications, including a chief. <B> alarms at other locations require an electrician. The pattern variable loc is bound to the value of the location field of <B> alarms. Finally, all other alarms require a chief (e.g. for inspecting the situation).

The axiomatization of this definition is straightforward of course, it just yields one axiom. However, the interesting aspect of the definition is the cases expression in the function body, which is translated to nested let and if expressions:

```
AlarmQualifications_defn
  "alarm : Alarm ==>
   AlarmQualifications@(alarm) =
   (let tf21 = alarm in
    if exists dc22 : Location & mk_Alarm(<A>,dc22) = tf21 then
      {<Mech>}
    elseif exists loc : Location & mk_Alarm(<B>,loc) = tf23 then
      let loc = tf23.loc in
      if loc in set {<P1>, <P3>, <P7>} then
        {<Chief>, <Elec>, <Chem>}
      else {<Elec>}
    else {<Chief>})"
```

The variables tf are generated automatically during the transformation. Again, as discussed above, a type judgement for the function body should strictly speaking be included in the hypotheses.

We can now define the safety requirement, which was stated in the invariant of Plant, using the function:

```
QualificationsOK: Alarm * set of Expert -> bool
QualificationsOK(alarm,exs) ==
  let reqquali = AlarmQualifications(alarm) in
    forall quali in set reqquali &
      exists ex in set exs & quali in set ex.quali
```

The axiomatization is obvious:

```
QualificationsOK_defn
  "[| alarm : Alarm; exs : set of Expert |] ==>
   QualificationsOK@(alarm, exs) =
   (let reqquali = AlarmQualifications@(alarm) in
    forall quali in set reqquali &
      exists ex in set exs & quali in set ex.quali)"
```

Again, this is in principle a derived axiom.

The function definitions above are all stated in the explicit style, which means that an algorithm is given for calculating the results. VDM-SL also supports an implicit style, where the result is just specified by a postcondition. The following function definition is in the implicit style:

```
Page(a:Alarm,per:Period,plant:Plant) r:set of Expert
pre per in set dom plant.plan and
    a in set plant.alarms
post r subset plant.plan(per) and
     AlarmQualifications(a) subset
       dunion {quali | mk_Expert(-,quali) in set r};
```

This function uses the plan of a plant to page a set of experts for a given alarm and period. A valid implementation of this function could just calculate all experts on duty for a certain period, regardless of the alarm, but a better implementation would probably try to minimize the set in some way.

Implicit function definitions yield four axioms, two defining the preconditions and postcondition respectively, and one definition and one formation axiom for the new function:

```
pre_Page_defn
  "pre_Page@(a, per, plant) ==
   per in set dom plant.plan and a in set plant.alarms"

post_Page_defn
  "post_Page@(a, per, plant, r) ==
   r subset plant.plan@(per) and
   AlarmQualifications@(a) subset
   dunion {let quali = tf31.quali in quali | tf31 in set r &
    exists quali : set of Qualification, dc32 : ExpertId &
      mk_Expert(dc32, quali) = tf31 and true}"

Page_defn
  "[| a : Alarm; per : Period; plant : Plant;
      pre_Page@(a, per, plant) |] ==>
   post_Page@(a, per, plant, Page@(a, per, plant))"

Page_form
  "[| a : Alarm; per : Period; plant : Plant;
      pre_Page@(a, per, plant) |] ==>
   Page@(a, per, plant) : set of Expert"
```

Note that the body of the postcondition is expanded slightly in order to treat the pattern in the set comprehension expression. The Toolbox parser inserts `true` when there is no restriction predicate in a comprehension expression. The Toolbox type checker ensures that the third axiom makes sense by generating a satisfiability proof obligation. Hence, also axioms for implicit functions are generated on the assumption that specifications are type correct.

7.8 Future Work

Although the current system can be used to reason about VDM-SL specifications, it can clearly be improved in many ways. Some future work and improvements are discussed briefly below.

Proof Rules

The proof system still needs work to be truly practical. Many of the rules for the different VDM-SL datatypes seem quite ad hoc and more work is needed here. Although often ignored, it is a major task to develop well-organized and powerful proof rules for datatypes like those of VDM-SL. Moreover, such proof rules should be organized in rule sets for the proof search tactics (if possible). This is also a major and important task.

Tactics

The current version of VDM-LPF does not provide a simplifier. Unfortunately it does not seem easy to instantiate Isabelle's generic simplifier to support reasoning about equality in VDM-LPF, since in order to do this we must justify that we can identify the VDM-LPF object-level equality with Isabelle's meta-level equality. If we cannot use Isabelle's simplifier, a special-purpose VDM-LPF simplifier should be constructed.

In addition, it might be useful to construct a number of special purpose tactics, for example, for reasoning about arithmetic and for proving trivial type conditions, which tend to clutter up proofs.

Compatibility

VDM-LPF is meant to be used to reason about VDM-SL and to be integrated with existing tools. A natural question is whether or not all these are compatible, and if not, what should then be changed. For example, is the proof system used sound with respect to the ISO standard (static and dynamic) semantics of VDM-SL? Although no inconsistencies have been found in the proof system itself, preliminary investigations suggest that there might be a problem with compatibility. For example, it is currently possible to derive `true or 1` as a theorem in the proof system, but according to the semantics of VDM-SL this is undefined. Since the semantics of VDM-SL is defined by an ISO standard it is perhaps most tempting to try to modify the proof system to exclude theorems as the one above. However, the result might be a proof system which is not suited for practical reasoning. This compatibility issue requires further attention.

Natural Deduction versus Sequent Calculus

The current instantiation is based on a natural deduction formulation of LPF. However, there are indications suggesting that a sequent calculus formulation of LPF might be a better foundation, in particular for automating proofs.

Status of the Current Prototype

The current prototype does not completely integrate the Toolbox and the theorem prover Isabelle, since the proof manager and the graphical user interface have not been implemented. However, it does provide the bits and pieces to support the integration. One shell script generates and transforms axioms, another parses, reverse transforms and pretty-prints Isabelle proof states. Proofs are presently conducted by interacting with Isabelle directly, which is feasible since the Isabelle instantiation supports a subset of the ISO standard for VDM-SL ASCII notation.

By exploiting transformations, we are able to treat essentially the full functional subset of VDM-SL. We have not considered, for instance, constructs like let-be-such-that expressions and union patterns, whose underdetermined semantics destroys reflexivity of equality [15]. As in Mural, we can treat state definitions and implicit operations. However, we have not yet considered explicit operations and statements, which form an imperative subset of VDM-SL, but already existing work on formalizing Hoare logic in the HOL theorem prover may be useful here [1]. Features of VDM-SL like exception handling have not been considered either.

7.9 Conclusion

In this chapter we have described the implementation of some central aspects of a proof support tool for VDM-SL based on Isabelle and how this tool has been integrated with the IFAD VDM-SL Toolbox. In particular we have illustrated the formalization of existing LPF proof rules in Isabelle and new facilities for automating proof search, which automatically proved essentially all of the 120 propositional and predicate logic derived rules listed in [6]. We feel that our experiments have shown Isabelle to be a very useful tool for quickly building specialized proof support for new and even non-standard logics such as VDM-LPF. Moreover, Isabelle is relatively easy to adapt to VDM-SL and to integrate with the IFAD VDM-SL Toolbox.

VDM-LPF supports only a subset of VDM-SL, and it is a major challenge to build proof support for the full VDM-SL standard. The language was designed for writing large specifications in industry, and this is reflected in both its syntax and its data types. On the syntax side, it supports pattern matching in for example let and quantifier expressions, and it has constructs such as cases expressions, again with patterns, which are difficult to represent in a theorem prover. On the data type side, it has non-disjoint unions, record types with postfix field selection, and arbitrary-length tuples that are not equivalent to nested pairs. Moreover, the underlying logic

of VDM-SL is the non-classical three-valued Logic of Partial Functions, which makes traditional classical approaches to, for example, proof search infeasible.

We are able to handle difficult constructs of VDM-SL by transforming these to expanded expressions in the VDM-LPF subset. However, the user needs never realize the transformations while writing proofs, since we can reverse transformations and provide a collection of derived proof rules which mimic the original expressions, though these actually work on expanded expressions. The transformations are dependent on the abstract syntax tree representation of expressions in the Toolbox, and would not be possible in Isabelle.

As a further consequence of the fact that the Isabelle instantiation supports the VDM-SL standard, we can use the Toolbox parser to read output from Isabelle. It has only been slightly modified to read proof rules, type judgements and subtypes, which are not part of the VDM-SL standard. Similarly, an ASCII pretty-printer for VDM-SL can be used to print abstract syntax both after it has been transformed to a subset understood by Isabelle and after Isabelle output has been parsed and reverse transformed to the original abstract syntax. Hence, no major changes or additions were needed to the IFAD Toolbox, in order to build the present prototype of an integrated system. Finally, as another consequence, the Isabelle instantiation can be used directly to reason in a subset of VDM-SL.

A main feature of the integrated CASE and theorem proving tool is that testing and proof for specification validation can be employed at the same time. All facilities of the IFAD VDM-SL Toolbox are available while conducting proofs. Moreover, the user always works in the notation provided by the Toolbox and is not limited by restrictions on syntax imposed by the proof component. Furthermore, it is possible to support industrial requirements like proof management, automation and version control, which typically are not well-addressed in theorem provers, outside the theorem prover in the proof support tool.

A generic framework like Isabelle allows quick implementation of powerful theorem provers through reuse. However, we have seen some limitations to reuse when dealing with a three-valued logic like VDM-LPF. In particular, the generic simplifier and classical reasoning package appear not to be easy to use with VDM-LPF. However, in our implementation of a special purpose reasoning package we were able to reuse many idea and even code from the existing package.

Acknowledgments

We would like to thank Peter Gorm Larsen for useful discussions concerning this work. Bernhard Aichernig and Peter Gorm Larsen commented on early drafts of this chapter. The work was financially supported by the Danish Research Councils.

7.10 Bibliography

[1] S. Agerholm. Mechanizing program verification in HOL. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*. IEEE Computer Society Press, 1992. A full version is in Technical Report IR-111, University of Aarhus, Department of Computer Science, Denmark.

[2] S. Agerholm. Translating specifications in VDM-SL to PVS. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[3] S. Agerholm and J. Frost. An Isabelle-based theorem prover for VDM-SL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, LNCS. Springer-Verlag, August 1997.

[4] S. Agerholm and J. Frost. Towards an integrated CASE and theorem proving tool for VDM-SL. In *FME'97*, LNCS. Springer-Verlag, September 1997.

[5] B. Aichernig and P. G. Larsen. A proof obligation generator for VDM-SL. In *FME'97*, LNCS. Springer-Verlag, September 1997.

[6] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994.

[7] J. H. Cheng. A logic for partial functions. Ph.D. Thesis UMCS-86-7-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, 1986.

[8] R. Elmstrøm, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL Toolbox: A practical approach to formal specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.

[9] J. Fitzgerald and P. G. Larsen. *Modelling Systems: Practical Tools and Techniques*. Camdridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1997. To appear.

[10] J. Fitzgerald, P. G. Larsen, T. Brookes, and M. Green. *Applications of Formal Methods, edited by M.G. Hinchey and J.P. Bowen*, chapter 14. Developing a Security-critical System using Formal and Convential Methods, pages 333–356. Prentice-Hall International Series in Computer Science, 1995.

[11] IFAD World Wide Web page. `http://www.ifad.dk`.

[12] C. B. Jones. *Systematic Software Development using VDM*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1986.

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

[14] P. G. Larsen, J. Fitzgerald, and T. Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.

[15] P. G. Larsen and B. S. Hansen. Semantics for underdetermined expressions. *Formal Aspects of Computing*, 8(1):47–66, January 1996.

[16] P. G. Larsen, B. S. Hansen, et al. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language. International Standard, ISO/IEC 13817-1, December 1996.

[17] P. Mukherjee. Computer-aided validation of formal specifications. *Software Engineering Journal*, pages 133–140, July 1995.

[18] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[19] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 199 – 208, Atlanta, Georgia, June 1988.

[20] PVS World Wide Web page. `http://www.csl.sri.com/pvs/overview.html`.

7.11 VDM-SL Syntax in Isabelle

This appendix describes the formalization of VDM-SL syntax in Isabelle, using an abstract and a concrete syntax and translations between them.

Abstract Syntax

We first provide a flavour of the core abstract syntax which is used internally in Isabelle for reasoning. It is a higher-order abstract syntax [19] in order to ensure smooth and elegant handling of bound variables. This means that object-level variables are identified with meta-level variables of a particular type. Meta-abstraction is used to represent variable binding, and substitution for bound variables is expressed using meta-function application. Isabelle's β-conversion handles the variable capturing problem. Moreover, typical side-conditions involving free variables are handled using universal meta-quantification and meta-application.

The abstract syntax has two central categories, one of expressions and one of types. Consequently Isabelle's typed lambda-calculus is extended with two new types `ex` and `ty` respectively. These types are logical in the sense that they are meant to be reasoned about, and new constants of these types are equipped with a standard prefix syntax.

Expressions and types of VDM-SL are represented in Isabelle using constants with result type `ex` and `ty`. Some examples of such constants are

```
not'    :: ex => ex
forall' :: [ty,ex => ex] => ex
eq'     :: [ex,ex] => ex
natty'  :: ty
succ'   :: ex => ex
```

These constants correspond to negation, universal quantification, equality, the type of natural numbers and the successor function. The constant for universal quantification is an example of a higher-order constant because it takes a function as its second argument. Using such constants as above it is possible to write expressions in a simple prefix form in the abstract syntax. The constants are primed in order to distinguish them from constants of the concrete syntax. The following example is a boolean expression which states that adding one is not the identity function for any natural number:

```
forall'(natty',%x.not'(eq'(x,succ'(x))))
```

This example illustrates how meta-abstraction `%` is used to express variable binding at the object-level.

Concrete Syntax

The purpose of the concrete syntax is to provide a user-friendly and familiar notation for presentation, while the role of the abstract syntax presented above was to support reasoning internally in Isabelle. The concrete syntax is based on the ISO standard of the VDM-SL ASCII notation [16]. This makes Isabelle/VDM-LPF relatively easy to use as a stand-alone tool for people who have experience with VDM-SL (and proof). Furthermore, it provides a standardized text-based format for exchanging data with other software components, such as the IFAD VDM-SL Toolbox.

The VDM-SL syntax standard is expressed as a context free grammar with additional operator precedence rules to remove ambiguities. The concrete syntax of VDM-LPF is expressed as a priority grammar, i.e as a grammar where the nonterminal symbols are decorated with integer priorities [18]. This priority grammar is constructed by a systematic and fairly straightforward translation of productions and operator precedence rules into priority grammar productions. In most cases the base form of the priority grammar productions comes directly from the corresponding production in the VDM-SL grammar, while the priorities of the production are constructed from the corresponding operator precedences. Some simple examples of such priority grammar productions are:

$$\begin{aligned} \mathrm{ex}^{(250)} &\leftarrow \texttt{not}\ \mathrm{ex}^{(250)} \\ \mathrm{ex}^{(310)} &\leftarrow \mathrm{ex}^{(310)} = \mathrm{ex}^{(311)} \end{aligned}$$

The structure of the above productions matches the corresponding declarations of the abstract syntax. Consequently, in such cases the concrete syntax is implemented

in Isabelle simply by adding a syntax annotation to the relevant constant declaration of the abstract syntax. The constant declarations corresponding to the two productions above are

```
not' :: ex => ex  ("(2not _/)" [250] 250)
eq'  :: [ex,ex] => ex  ("(_ = /_)" [310,311] 310)
```

where the types correspond to nonterminals of the productions. The syntax annotation in brackets consists of two parts: a quoted mixfix template followed by an optional priority part. The mixfix template describes the terminals and contains other printing and parsing directives (see [18]).

However, not all of the concrete syntax can be handled by adding syntax annotations to the constant declarations for the abstract syntax. In cases such as multiple binding quantifiers, set comprehensions, if-then-elseif expressions, enumerated sequences, etc., the structure of the concrete syntax differs from that of the abstract syntax. Such situations are handled using separate syntax declarations which declare a special kind of constants. These constants only serve a syntactic purpose and are never used internally for reasoning. The syntax declarations below are those needed for multiple binder universal quantifications:

```
""      :: tbind => tbinds  ("_")
tbinds_ :: [tbind,tbinds] => tbinds  ("(_,/ _)")
tbind_  :: [idt,ty] => tbind  ("(_ :/ _)")
forall_ :: [tbinds,ex] => ex  ("(2forall/ _ &/ _)" [100,100] 100)
```

Unlike when using syntax annotations, the relationship to the abstract syntax is not established automatically in such separate syntax declarations. Instead translations are defined to relate the abstract and the concrete syntax, as discussed in the following section.

Translations

A simple example of a translation is the expansion of the special notation for not equal <> in the concrete syntax to negated equality in the abstract syntax. This is implemented using Isabelle's macro mechanism. A macro is essentially a rewrite rule which works on Isabelle's abstract syntax trees. In this case the macro is

```
"x <> y" == "not x = y"
```

This translation means that not equal will behave just as a negated equality in proofs. Another deliberate consequence is that any negated equality will be presented using the more readable not equal notation. In other words the original formatting is not always retained, but instead the implementation tries to improve it whenever possible.

Another more complicated example is that of universal quantification with multiple type bindings. In VDM-LPF such a quantifier is viewed simply as a syntactic shorthand for a number of nested single binding quantifiers. The translation between the

concrete external first-order representation and the internal higher-order abstract representation is implemented using a mixture of macros and print translations. However, these are too complex to be included here. During parsing, the effect of these macros and translations is that concrete syntax such as

```
forall x:nat, y:nat & x = y
```

is automatically translated to the following abstract syntax:

```
forall'(nat,%x.forall'(nat,%y.eq'(x,y)))
```

Similarly, during printing, the abstract form is translated to the concrete form. Constants such as `tbind_` and `forall_` of the previous section do not occur explicitly here, since they are only used in the standard first-order syntax trees corresponding to the concrete syntax. However, they are used in the relevant macros and translations. Other variable binding constructs are handled in a similar fashion to universal quantification.

Index