Proving Consistency of VDM models using HOL

Sander D. Vermolen
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
s.d.vermolen@tudelft.nl

Jozef Hooman
Embedded Systems Institute
& Radboud University
Nijmegen
The Netherlands
jozef.hooman@esi.nl

Peter Gorm Larsen
Engineering College of Aarhus
Dalgas Avenue 2
Aarhus
Denmark
pgl@iha.dk

ABSTRACT

Although consistency of formal models is crucial, consistency proofs should not be a large burden to the user. Hence, it is important to have access to efficient proof support which is able to automate a large part of the consistency proofs. We have developed a tool that automatically translates a large subset of VDM and its associated proof obligations, which ensure model consistency, to the theorem prover HOL. In addition, powerful tactics have been constructed to discard most of the proof obligations automatically. The application of our approach to four case studies shows that a high degree of automation can be achieved.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

Keywords

Model consistency, verification, theorem proving, VDM, HOL

1. INTRODUCTION

The general aim of our work is to improve the usability of the formal method VDM in software engineering. VDM is a well established formal method with a long history [8, 16] and a strong record in industrial applications (see, e.g., [19, 21, 17, 18]). VDM models are expressed in a textual specification language called VDM-SL, which is a standard of the British Standards Institution and ISO [6]. This modeling language has a formal semantics and allows a wide range of analytical techniques.

Different VDM dialects, such as the object-oriented extension VDM++, are supported by industry-strength tools. These tools are combined into the tool VDMTools, which is currently owned and further developed by CSK [10, 14]. VDMTools includes code generation and round-trip engineering with UML. An open-source initiative called Overture [25] is being developed to allow research on innovative tool support, such as the work described in this paper which addresses tool support to automate consistency proofs.

To appear in the Proceedings of the 25th Symposium On Applied Computing (SAC'10), Technical Track on Software Verification and Testing, Sierre, Switzerland, 22-26 March 2010, ACM.

An important general problem of the use of formal models is that it is usually easy to construct inconsistent models, which may lead to erroneous formal reasoning and faults in the final realization. Unfortunately, it can be difficult to detect inconsistencies or to prove internal consistency manually. Especially for large and complex models, the manual proof of consistency is a tedious and time consuming task. This also holds for VDM models and can be a blocking factor for large-scale applicability.

The statically decidable inconsistencies in a VDM model will be determined by the type checker. Remaining potential inconsistencies will be located by the integrity checker of VDMTools [5]. Whenever a potential inconsistency is found, the integrity checker generates a predicate, a so-called proof obligation, which is also expressed in VDM. The collection of proof obligations is complete, that is, if they are all valid then execution of the model according to the dynamic semantics will not lead to any run-time errors. In terms of the dynamic semantics from the ISO VDM-SL standard, this means that at least one mathematical model will exist [20].

The goal of our work is to reduce the amount and the complexity of manual consistency proofs by discharging, as far as possible, proof obligations generated by the integrity examiner of VDMTools automatically. To this end, we automatically translate both the obligations and the model (which is typically needed to prove obligations) to a general-purpose theorem prover. Currently, the translation is restricted to the functional subset of VDM, that is, functions can only be expressed either implicitly using preconditions and postconditions or explicitly with an expression calculating the result based on the input provided. We also exclude mutually recursive functions, for reasons explained in Section 4.1.

Relevant preparatory research has been done on the translation of a subset of VDM-SL to theorem provers such as PVS [2, 22]. These papers describe the manual translation of case studies and they do not address automatic proof of obligations. A deep embedding of VDM-SL in Isabelle has been defined in [3]. Our main source of inspiration is the Prosper project [12] in which research has been conducted on a general open proof tool architecture for incorporating formal verification into industrial tools. As a case study, they have investigated the automatic translation of VDM-SL models into the HOL theorem prover [7]. Since we build upon the main ideas of the Prosper project, we have also used the HOL prover, although we moved from the older version HOL98 to the latest version called HOL4.

In contrast with the related work described above, our work has lead to concrete tool support that can be used in the Overture tool set and which is freely available from the project web site[1]. An important part of our work is a translation of VDM to HOL that deals with the incompatibilities between these two domains. The

[1]http://www.overturetool.org/twiki/bin/view/Main/AutomaticProof

main differences concern

- *partiality*, the logic of partial functions on which VDM is based, as opposed to the 2-valued logic of HOL;
- *type invariants*, which are easily defined in VDM whereas HOL has limited support for defining them automatically;
- *implicit functions*, defined by preconditions and postconditions, which are absent in HOL;
- *patterns*, which are used extensively in VDM but hardly supported in HOL.

In addition, we have built a powerful proof tool for the VDM proof obligations based on HOL. We use a classification of proof obligations to select suitable strategies for proving them. In contrast to earlier approaches, our focus is on fast proofs and on discovering a high number of successful proofs automatically, using the existing HOL libraries as much as possible. Earlier work, such as the translation of the PROSPER project [12] and the embedding in [3], required a large amount of additional theorems since they did not use the HOL constructions and methods directly. In contrast, the result of our translation is a model that can be inserted into HOL directly and does not require any additional libraries or theories for reasoning about VDM concepts. We have applied our system to four case studies to assess the degree of automation, usefulness and usability with respect to performance.

The remainder of this paper is structured as follows. Section 2 contains a VDM example to demonstrate the main concepts of the language and the proof obligations. Our general approach is described in Section 3. Section 4 explains the translation from VDM to HOL in more detail. Our strategies to prove obligations automatically are discussed in Section 5. Concluding remarks can be found in Section 6.

2. VDM OVERVIEW

We illustrate the main concepts of VDM using an abstract version of the Mondex Electronic Purse case study [26, 1]. The model defines an electronic purse and a world containing several purses. Functions are available to increase or decrease balances of purses, to request information about purses or about the world and to verify transfers of money.

The VDM model of the *AbPurseFunctional* class shown below starts by defining three types. Type *AbPurse* is the Cartesian product of two fields, where field *lost* models money that has been subtracted from the purse by a failed transfer. Type *PurseId* defines identifiers using basic type token which provides tokens without any further structure. Type *AbWorld* is also the Cartesian product of two fields, where $\xrightarrow{m}$ denotes a finite function. Type *AbWorld* has an invariant which expresses that all purse identifiers in the domain of the mapping are *authentic*. Note that *mk-AbWorld* constructs an instance of type *AbWorld*.

class $AbPurseFunctional$
types
public $AbPurse :: balance : \mathbb{N}$
$lost : \mathbb{N}$;
public $PurseId = \mathsf{token}$;
public $AbWorld :: authentic : PurseId\text{-}\mathsf{set}$
$abPurses : PurseId \xrightarrow{m} AbPurse$
inv mk-$AbWorld\,(authentic, abPurses) \triangleq$
$\forall\, name \in \mathsf{dom}\ abPurses \cdot name \in authentic$

Next we define functions to return and to increase the values of the fields of an *AbPurse* instance.

functions
public $GetBal : AbPurse \tilde{\to} \mathbb{N}$
$GetBal\,(p) \triangleq$
$p.balance$;
public $GetLost : AbPurse \tilde{\to} \mathbb{N}$
$GetLost\,(p) \triangleq$
$p.lost$;
public $IncreaseBalance : AbPurse \times \mathbb{N} \tilde{\to} AbPurse$
$IncreaseBalance\,(p, val) \triangleq$
mk_$AbPurse\,(p.balance + val, p.lost)$;
public $IncreaseLost : AbPurse \times \mathbb{N} \to AbPurse$
$IncreaseLost\,(p, val) \triangleq$
mk_$AbPurse\,(p.balance, p.lost + val)$;

The function to reduce a balance by a given value has a precondition to ensure a non-negative balance. The total amount of money in a purse is defined by function *GetTotal* as the sum of its balance and lost values.

public $ReduceBalance : AbPurse \times \mathbb{N} \tilde{\to} AbPurse$
$ReduceBalance\,(p, val) \triangleq$
mk_$AbPurse\,(p.balance - val, p.lost)$
pre $p.balance \geq val$
public $GetTotal : AbPurse \tilde{\to} \mathbb{N}$
$GetTotal\,(p) \triangleq$
$p.balance + p.lost$;

Function *newAbWorld* creates a new *AbWorld* instance.

$newAbWorld : PurseId \xrightarrow{m} AbPurse \times PurseId\text{-}\mathsf{set} \tilde{\to} AbWorld$
$newAbWorld\,(purses, auth) \triangleq$
mk_$AbWorld\,(auth, purses)$
pre $\mathsf{dom}\ purses \subseteq auth$

Function *TransferOk* models the transfer of a value from one purses' balance to the balance of another purse. The precondition expresses that the transfer is between two different purses, both purses exist and are authentic and there must be sufficient funds in the 'from' purse *frm*. In the definition of the function, symbol † denotes the override of a function for a particular value (here *frm*) of its domain.

In contrast to the purely *explicit* function definitions above (where an expression defines the result), we provide an additional *implicit* definition for *TransferOk* by means of a postcondition. In this case, the postcondition states that the total (balance + lost) of both purses involved in the transfer must be unchanged, no value must be created, and all other purses remain unchanged. The identifier *RESULT* in the postcondition is the standard way in VDM to denote the result value of the function. Note that such a combined definition leads to a proof obligation which requires that the explicit definition satisfies the postcondition.

```
public  TransferOk : AbWorld × PurseId × PurseId × ℕ~→
                     AbWorld
        TransferOk (wrld, frm, too, val) ≜
          (let newFrm =
                 ReduceBalance (wrld.abPurses (frm), val),
               newTo =
                 IncreaseBalance (wrld.abPurses (too), val) in
           mk_AbWorld (wrld.authentic,
                       wrld.abPurses †
                       {frm ↦ newFrm, too ↦ newTo}))
        pre  frm ≠ too ∧
             frm ∈ dom wrld.abPurses ∧
             too ∈ dom wrld.abPurses ∧
             (GetBal (wrld.abPurses (frm)) ≥ val)
        post (GetTotal (RESULT.abPurses (frm)) +
              GetTotal (RESULT.abPurses (too)) =
              (GetTotal (wrld.abPurses (frm)) +
               GetTotal (wrld.abPurses (too))) ∧
              (GetBal (RESULT.abPurses (frm)) +
               GetBal (RESULT.abPurses (too))) =
              (GetBal (wrld.abPurses (frm)) +
               GetBal (wrld.abPurses (too))) ∧
              ∀ nm ∈
              (dom RESULT.abPurses) \ {frm, too} ·
                  (GetBal (wrld.abPurses (nm)) =
                   GetBal (RESULT.abPurses (nm))) ∧
                  (GetLost (wrld.abPurses (nm)) =
                   GetLost (RESULT.abPurses (nm))))
```

Function *TransferLost* models a failed transfer where a value is moved from one purses' balance to the lost variable of the same purse. It is defined by a purely *implicit* function definition. The precondition requires that the 'from' purse must exist and be authentic and contains sufficient funds. The postcondition expresses that the total (balance + lost) of the purse must be unchanged, no value must be created, (the 'before' balance must be greater or equal to the 'after' balance), and all other purses remain unchanged.

```
public  TransferLost (wrld : AbWorld, frm : PurseId,
                      -:PurseId, val:ℕ) RESULT:AbWorld
        pre  frm ∈ dom wrld.abPurses ∧
             GetBal (wrld.abPurses (frm)) ≥ val
        post GetTotal (RESULT.abPurses (frm)) =
             GetTotal (wrld.abPurses (frm)) ∧
             GetBal (wrld.abPurses (frm)) ≥
             GetBal (RESULT.abPurses (frm)) ∧
             ∀ nm ∈
             (dom RESULT.abPurses) \ {frm} ·
                 GetBal (wrld.abPurses (nm)) =
                 GetBal (RESULT.abPurses (nm)) ∧
                 GetLost (wrld.abPurses (nm)) =
                 GetLost (RESULT.abPurses (nm))
```

Instead of identifiers which denote variables, VDM also supports many types of patterns, such as the don't care pattern "-" in the parameter list of *TransferLost* above. In this case, the don't care pattern is used to conform to the fixed interface of the Mondex specification; the parameter is irrelevant at this level of abstraction, but is used in a more refined variant.

In addition to the Mondex case, we have evaluated our proof strategies on a number of other case studies, such as an alarm system of a chemical plant introduced in [13], a memory allocation model, a tracking system of containers in a nuclear plant and the control of a propulsive backpack system introduced in [23, 4]. The cases have been selected because they are realistic and provide a considerable number of proof obligations (62 in total) that are relatively hard to prove.

2.1 Proof Obligations

The integrity checker of VDMTools distinguishes many types of proof obligations. Many of these types can be proved in similar ways and do not require their own mechanisms. Hence, similar to [5], we distinguish four groups of proof obligations:

- *Domain checking:* due to the use of partial functions and partial operators.
- *Subtype checking:* due to the use of subtypes (in particular, type invariants).
- *Satisfiability of implicit definitions:* due to the use of postconditions.
- *Termination:* due to the use of recursive functions.

In this paper, we put little emphasis on termination, since no new tactics have been developed for this class. The next subsections show examples of the first three classes of proof obligations.

2.1.1 Domain checking

The domain checking proof obligations are generated as a result of the use of partial operators and partial functions. They are the most common type of obligations and usually also the easiest to prove.

In the Mondex case study, the domain checking proof obligations express, for instance, that whenever the function *ReduceBalance* is called, its precondition has to be satisfied. As an example, inside the function *TransferOk* there is an application of the *ReduceBalance* function leading to the proof obligation:

$$\forall\, wrld : AbWorld, frm : PurseId, too : PurseId, val : \mathbb{N} \cdot \\ \mathsf{pre\_}TransferOk\,(wrld, frm, too, val) \;\Rightarrow \\ \mathsf{pre\_}ReduceBalance\,(wrld.abPurses\,(frm), val)$$

2.1.2 Subtype checking

The subtype proof obligations mainly result from the use of invariants. To illustrate this category, recall that in the Mondex case the definition of function *newAbWorld* constructs a new *AbWorld*. Since type *AbWorld* has an invariant, a proof obligation is generated to verify that this invariant is satisfied:

$$\forall\, purses : PurseId \xrightarrow{m} AbPurse, auth : PurseId\text{-}\mathsf{set} \cdot \\ \mathsf{inv\_}AbWorld\,(\mathsf{mk\_}AbWorld\,(auth, purses))$$

Using the definitions of the model, i.e., the definitions of the *AbWorld* type and the *newAbWorld* function, this can be rewritten to:

$$\forall\, purses : PurseId \xrightarrow{m} AbPurse, auth : PurseId\text{-}\mathsf{set} \cdot \\ \forall\, name \in \mathsf{dom}\ purses \cdot name \in auth$$

Note that this condition does not hold. As a solution, we add the invariant of the *AbWorld* type as a precondition to the *newAbWorld* function. This invariant is then also included in the changed proof obligation, which makes its proof trivial.

2.1.3 Satisfiability

The satisfiability proof obligations are caused by the use of implicit function definitions. As an example of a satisfiability problem

in the Mondex case, consider the proof obligation that is generated to ensure that the postcondition of the *TransferLost* function is satisfiable. Note that the don't care parameter of the function is removed in the proof obligation and the result of the function is represented by the last parameter *r* of the postcondition.

$$\begin{aligned}&\forall\, wrld : AbWorld, frm : PurseId, val : \mathbb{N} \cdot \\ &\quad \mathsf{pre\text{-}}TransferLost\,(wrld, frm, val) \;\Rightarrow \\ &\quad \exists\, r : AbWorld \cdot \mathsf{post\text{-}}TransferLost\,(wrld, frm, val, r)\end{aligned}$$

3. APPROACH

To be able to build upon results of the PROSPER project [12], we use the HOL proof assistant to discharge proof obligations. HOL is a generic theorem prover which is highly suitable for reasoning about expressions using a large set of readily available theories. The HOL system is designed for interactive as well as automated theorem proving. It uses Higher Order Logic (hence HOL) based on typed lambda calculus. HOL distinguishes two levels: the meta level expressed in the Meta Language (ML) and an object level expressed by terms.

In order to reason about a VDM model and its proof obligations in the HOL theorem prover, we connect the two domains by a semantics preserving translation. The main difference with the PROSPER approach concerns the reliance of the PROSPER translation on a large library of VDM supporting definitions and the use of an additional set of custom HOL libraries to support the translation result. One disadvantage of this approach is the need for libraries in two domains to construct the translation. Another, more important disadvantage is that much functionality that is built into HOL can no longer be used directly. The main focus of our work has been to use as much of HOL's possibilities as possible. The VDM to HOL translation consists of three steps:

1. The first step takes a concrete VDM model which is parsed into an abstract VDM representation using the parser of the Overture tool set [25].
2. The second step translates the abstract VDM model into an abstract HOL model and is described in Section 4. It consists of smaller steps with, e.g., transformations from VDM to VDM and from one intermediate format to another.
3. The last step converts the abstract HOL model into a concrete HOL model, i.e., lines of HOL code that can be used directly in the HOL proof system to reason about the model.

The translation is based on two assumptions on the input model, namely syntactic correctness as defined in [27] and static type correctness, as defined in [9, 11] for VDM-SL. The first is verified by the Overture parser, in the first step above. The second can be verified on beforehand by the type checker of Overture or the type checker of VDMTools.

4. TRANSLATING VDM TO HOL

In this section, we first discuss partiality in Section 4.1. The translation of a VDM model is defined using three translations that call each other recursively:

- a type translation, which takes a VDM type as input and produces a HOL type,
- an expression translation, which translates a VDM expression to a HOL term, and
- a definition translation, which takes a VDM definition and yields a Meta Language (ML) statement.

These translations are discussed in Subsections 4.2 through 4.4. Invariants are discussed separately in Section 4.5, since they are treated using a rewriting in terms of an intermediate format. Many of the type and expression translations are easily constructed and will therefore not be discussed. We focus on a detailed discussion of a small selection of the more complex constructs.

4.1 Partiality

Observe that there is an incompatibility between the logic of partial functions (LPF) on which VDM is based [15] and two-valued logic of HOL. To prevent the application of a function outside its domain, which would lead to undefined results in HOL, we use the proof obligations themselves, since they imply the existence of an evaluation in two-valued logic [5]. In general, we know from the definition of the proof obligations that every time a partial function or operator is being used, a proof obligation is generated stating that its precondition should hold, thus ensuring the existence of an evaluation in two-valued logic. This holds for the usage of partial functions in function bodies, as well as the usage of partial functions in preconditions, or any other location in the specification. However, a problem might occur if a proof obligation is generated that is potentially partial itself. This can happen when using partial preconditions. Consider, for example the *GetBal* function which, given a purse, yields its balance. Suppose that we would define this function based on a purse id, instead of a purse itself:

$$\begin{aligned}&\mathsf{public}\quad GetBal : AbWorld \times PurseId \xrightarrow{\sim} \mathbb{N} \\ &\qquad GetBal\,(wrld, id) \triangleq \\ &\qquad\quad wrld.abPurses\,(id).balance \\ &\qquad \mathsf{pre}\;\; id \in \mathsf{dom}\; wrld.abPurses\end{aligned}$$

The additional parameter *wrld* is required to lookup the given purse id. A precondition is required to ensure that the id is actually associated to a purse. The precondition now makes the function partial. Using this altered *GetBal* definition, we can simplify the precondition of *TransferOk* to:

$$\begin{aligned}&\mathsf{public}\quad TransferOk\,(wrld, frm, too, val) \triangleq \\ &\qquad\quad ... \\ &\qquad \mathsf{pre}\;\; frm \neq too \wedge \\ &\qquad\qquad frm \in \mathsf{dom}\; wrld.abPurses \wedge \\ &\qquad\qquad too \in \mathsf{dom}\; wrld.abPurses \wedge \\ &\qquad\qquad (GetBal\,(wrld, frm) \geq val) \\ &\qquad \mathsf{post}\;\ldots\;;\end{aligned}$$

If we now apply *TransferOk* inside another function *f*, a proof obligation (*po1*) is generated stating that the precondition of *TransferOk* should hold to guarantee that f can be evaluated in two-valued logic. Yet, this proof obligation is itself partial as the partial function *GetBal* is applied inside the precondition. However, an additional proof obligation (*po2*) stating that the pre-condition of *GetBal* should hold inside the precondition of *TransferOk*, will guarantee a two-valued result of *po1*.

The example above illustrates a dependency of proof obligations caused by a dependency of preconditions. These dependencies do not pose a threat unless they are circular. This would have been the case if, although unlikely, the precondition of *GetBal* would have refered to *TransferOk*. In that case our translation fails and hence we have excluded mutual recursive functions from our VDM subset in Section 1.

4.2 Type translation

Since most VDM types have a counterpart in HOL, their trans-

lation consists of basic rewrites. As an example, consider a finite VDM mapping from type *Td* to type *Tr*, written as map *Td* to *Tr* (or $Td \xrightarrow{m} Tr$ using the mathematical notion of Section 2). This is translated to a finite mapping in HOL as follows:

$< Td \xrightarrow{m} Tr >=$ `(<Td> |-> <Tr>)`,

where the brackets $< \ldots >$ denote the type translation, which is applied recursively to translate *Td* and *Tr*.

Similarly, most values have a counterpart in HOL. Since HOL does not directly support map enumerations, the VDM enumeration is translated to a repeated map update (`|+`) starting with the empty map called `FEMPTY`. For instance, the VDM map enumeration $\{1 \mapsto 2, 2 \mapsto 3\}$, which maps 1 to 2 and 2 to 3, is translated to `FEMPTY |+ (1, 2) |+ (2, 3)`.

Having defined a translation for types, we face a more difficult problem, namely the usage of types in a model. For instance, in the model of the Mondex case study presented in Section 2, the type token – used to define type *PurseId* – is simply translated to HOL type `ind` (short for induction), which contains any value. On the other hand, translating the usage of *PurseId*, e.g., in function *newAbWorld*, requires a type definition mechanism in HOL. Unfortunately, defining types in HOL is not as simple as it is in VDM. When it comes to this part of the translation, we distinguish three possible strategies:

(1) Advanced type definitions.

HOL has a built-in function '`hol_datatype`' that defines advanced data types for the user. It requires the description of the type and a type name as input (in a syntax specific for the function) and produces the type definition as a result, as well as several proved theorems that ease the use of it (e.g. an induction theorem) [7]. `hol_datatype` primarily handles records, tree structures and enumerations.

This method is simple to use, but it is only meant to deal with data types, meaning that it is unable to handle simple type definitions such as the synonym type *PurseId* above. Furthermore, type invariants cannot be included in the definition.

(2) Subtyping.

A more powerful type definition method in HOL is the use of an existing type. Instead of giving an explicit description of the values belonging to a certain type, the values are described by restricting the value base of an existing type using an invariant. The appropriate abstraction and representation functions to obtain and use values of the type are automatically defined by HOL. A disadvantage of this method, is that it requires an explicit proof that the type being defined is not empty.

(3) Omitting type definitions.

Instead of using one of HOL's type definition methods, we can also use a VDM-specific approach. Due to the amount of work, it is not advisable to write a new definition mechanism specific to VDM. However, there are two alternatives:

(3a) Meta variables Meta type variables exist in the Meta Language domain and can be used to store types temporarily. A similar approach using meta variables has been tried in [3] on variable binding.

(3b) By translation By making more extensive use of our translation, we can prevent the need for type definitions in HOL. During translation all synonym type definitions can be recorded and all synonym type occurrences can be replaced by their defining type. For instance, we could replace occurrences of type *PurseId* by token. This approach is not applicable to more advanced type definitions.

Our first attempt to translate VDM type definitions was by means of the second strategy, namely subtyping. This allowed for the incorporation of invariants and seamless integration into the model (no names had to be changed and no additional definitions were required). Unfortunately, non-emptiness of types is difficult to prove automatically, and in general this cannot be solved efficiently. Giving our aim to automate proofs as much as possible, this strategy was rejected. Our current translation uses a combination of two other strategies:

1. For record types and quote types, we apply the advanced type definition method of strategy *(1)*. It introduces several theorems along with the definition (e.g., theorems supporting induction) automatically.

2. All other types (tuples, unions, enumerations, synonyms, etc.), are translated by strategy *(3b)*. Their definition is omitted in the final translation result and all type names belonging to this category are replaced by their definitions. This method is preferred over the use of meta variables as used in [3], since our approach avoids models at the HOL level being bound to a context on the meta level and hence provides more flexibility during proof attempts and the storage of these attempts.

4.3 Expression translation

A large part of the expression translation is straightforward. We therefore do not present the translation in detail, but only briefly discuss the translation of patterns.

The VDM language definition supports the use of patterns in many of the language constructs. Since HOL generally does not support patterns, all pattern occurrences in VDM models are rewritten. For example,

$\mathsf{let\ mk\text{-}}(x, y, 7) = someCoord\ \mathsf{in}\ \ldots$

indicates that the first field of tuple *someCoordinate* should be assigned to *x*, the second field to *y* and the third field should equal 7. This is rewritten to:

$$\begin{aligned}\mathsf{let}\ & x = someCoord.\#1,\\ & y = someCoord.\#2,\\ & 7 = someCoord.\#3\ \ \mathsf{in}\ \ldots\end{aligned}$$

Whereas the first two lines assign values to the variables x and y, the third line is an equality check instead of an assignment. When this check fails, the entire let expression cannot be executed and will therefore fail. Hence, the third part makes this expression partial. As explained in Section 4.1, a proof obligation is generated to avoid such a failure.

4.4 Definition translation

The VDM subset supported by the translation contains three types of definitions: definitions of implicit and explicit functions, and definitions of types. Definitions cannot be part of anything else but the model itself. Consequently, definitions are put at the highest level of a model. This also holds for HOL, in which definitions are expressed at the meta level. The translation therefore has ML as target domain. The translation of type definitions has already been described in Sect. 4.2, so we only focus on explicit and implicit function definitions here.

Explicit functions.

A VDM explicit function declaration which also has a precondition and a postcondition (as function *TransferOk* in Section 2) has the following layout:

$$
\begin{array}{l}
\mathit{functionName} : T_1 \times T_2 \times \ldots \times T_n \tilde{\rightarrow} T_{n+1} \\
\mathit{functionName}\,(p_1, p_2, \ldots, p_n) \triangleq \\
\quad E \\
\mathsf{pre}\ \mathit{Pr} \\
\mathsf{post}\ \mathit{Po}
\end{array}
$$

Ignoring language constructions that were described before (such as patterns), the first part of the translation has the following outline:

```
Define `functionName
      (<|p1|> : <T1>) ... (<|pn|> : <Tn>) =
         <|E|> : <Tn+1>`;
```

where <...> and <|...|> denote the type and expression translation, respectively. **Define** is a HOL function existing on the meta level, that eases the definition of functions. HOL will try to prove termination of the given function before it continues with the definition process. Observe that termination is one of the proof obligations required for a successful execution of a model, and hence it has to be proved anyway. Section 5.3 will discuss the proof of termination.

The preconditions and postconditions of the function (*Pr* and *Po*) are defined separately as boolean functions. By the definition of the proof obligations, their names are predefined to be 'pre_functionName' and 'post_functionName'.

Implicit functions.

The general layout of an implicit function definition is:

$$
\begin{array}{l}
\mathit{functionName}\,(p_1 : T_1, \ldots, p_n : T_n)\ \mathit{RESULT} : T_{n+1} \\
\mathsf{pre}\ \mathit{Pr} \\
\mathsf{post}\ \mathit{Po}
\end{array}
$$

Implicit function definitions have a lot in common with explicit ones. Therefore, their translation uses the translation of the explicit function definitions. The actual translation of implicit functions includes a VDM to VDM rewrite step in which an exact expression is given that defines the function result. This is done by means of the 'let be such that' expression in VDM as follows:

$$
\begin{array}{l}
\mathit{functionName} : T_1 \times T_2 \times \ldots \times T_n \tilde{\rightarrow} T_{n+1} \\
\mathit{functionName}\,(p_1, p_2, \ldots, p_n) \triangleq \\
\quad \mathsf{let}\ \ \mathit{RESULT} : T_{n+1}\ \mathsf{be\ st}\ \ \mathit{Po}\ \mathsf{in} \\
\quad \mathit{RESULT} \\
\mathsf{pre}\ \mathit{Pr}
\end{array}
$$

The let expression defines that the local variable *RESULT* satisfies *Po* and is of type T_{n+1}. Note that in most proof attempts, the 'let be such that' expression with a certain postcondition will be replaced, using a few simple rewrites, by a new (semantically equivalent) proof goal containing merely the postcondition.

Comparing our approach to the one used by Agerholm in his translation of VDM-SL to PVS [2], one can observe a similar result by using a different method. Agerholm uses an uninterpreted constant function, of which the resulting values satisfy the postcondition if the precondition holds. Of particular interest is the behaviour if the precondition is not satisfied, which in both cases is undetermined. Both approaches thereby depend on the proof obligations for not using the function body in those scenarios, as explained in Section 4.1.

4.5 Invariant translation

In the previous sections, we have ignored type invariants. Including them in the type definition would have been the fastest approach, but this has not been adopted because it would require non-emptiness proofs that would obstruct the automation. Instead, we insert the invariant at all required places by means of a translation from one intermediate format to another.

Observe that whenever a type is being used in the VDM model, its invariant (if present) has to be included in the HOL model, to ensure that the semantics of both models are equivalent. The number of places where types can occur in VDM models is limited. We will discuss each of them individually and show how to insert the invariant. Although the translation is defined in terms of an intermediate format, the implementation details are irrelevant and would obscure the main concepts. Hence we describe the main ideas in terms of logical formulas, using types T, T_1, and T_2, with invariants $\mathit{inv}T$, $\mathit{inv}T_1$, and $\mathit{inv}T_2$ imposed on their super types T', T_1', and T_2', respectively.

Quantifiers and Quantifier-like expressions.

Consider the occurrence of types with invariants in quantifiers. For a predicate P, a universal quantification $\forall x : T \cdot P(x)$ is rewritten to the equivalent expression $\forall x : T' \cdot \mathit{inv}T(x) \Rightarrow P(x)$. Similarly, $\exists\, x : T \cdot P(x)$ is rewritten to $\exists\, x : T' \cdot \mathit{inv}T(x) \wedge P(x)$.

In addition to the existential quantifier, there are several more translations that simply introduce the invariant by means of a conjunction, e.g., for the 'let be such that' expression.

Function definition.

Translating function definitions may involve several invariants, since they can carry more than one type. For instance, suppose the source of the translation is a function $f : T_1 \rightarrow T_2$, with $f(x) \triangleq E(x)$, pre Pre, and post Post. Then the translation includes both the source and the target invariants, leading to $f : T_1' \rightarrow T_2'$ with $f(x) \triangleq E(x)$, pre $\mathit{Pre} \wedge \mathit{inv}T_1(x)$, and post $\mathit{Post} \wedge \mathit{inv}T_2(\mathit{RESULT})$.

Type definition.

Finally, we consider types that are used in type definitions themselves. We distinguish two kinds of type definitions. The first is the regular type definition of a new type, say T_{new}, of the form: $T_{new} = T$ with inv $x = \mathit{inv}T_{new}$. Here x is a pattern identifier referring to an example element of the new type which can be used in the invariant predicate to express constraints. This will be translated to $T_{new} = T'$ with inv $x = \mathit{inv}T_{new} \wedge \mathit{inv}T(x)$.

The second is the definition of a record type, which has an arbitrary number of fields, of which one is of a non-primitive type T: $T_{new} = \ldots f_i : T \ldots$ with inv $x = \mathit{inv}T_{new}$. This is rewritten to $T_{new} = \ldots f_i : T' \ldots$ with inv $x = \mathit{inv}T_{new} \wedge \mathit{inv}T(x.f_i)$. This approach is also valid for record types with multiple fields of non-primitive types.

Observe that in both constructions $\mathit{inv}T$ is not the invariant of type T defined in the model, but the result of recursive application of the translation defined here.

5. PROVING PROOF OBLIGATIONS

Given the translation defined in the previous section, both the definitions from the VDM model and its proof obligations are translated to HOL. In this section, we discuss the automated proof of these obligations in HOL.

In general, a theorem in HOL is proved by applying inference rules to axioms or existing theorems that have been proved before. There are five primitive axioms and eight basic inference rules. An inference rule is a HOL function that yields a theorem. Finding a

proof is basically a backward search, starting at the goal, i.e., the theorem to be proved. Tactics are ML functions that can be applied to a goal and result in a list of subgoals and a justification function. This justification function will prove the goal using the subgoals based on inference. There are several tactics built into HOL that we can use directly. To increase the amount of automation in our domain, we have defined additional tactics that are able to discharge most of the VDM proof obligations. An important concern for the development of the tactics was to limit the time needed to prove each of the obligations. When evaluating our tactics, a proof attempt has never been allowed to take longer than 10 seconds.

In the next subsections, we discuss the proof of the obligations mentioned in Section 2.

5.1 Domain checking

To prove a domain checking proof obligation, such as the example in Section 2.1.1, it is rewritten first, for instance, using the definitions of the preconditions. In general, suitable rewriting is not obvious, since in some cases it is useful to rewrite all definitions, whereas in other cases it might be more useful to use earlier proved theorems about higher level functions.

Inspecting the domain checking proof obligations in case studies, it turns out that most of them consist of relatively simple basic logic. Consequently, many of these obligations can be proved by means of simple logical inference rules. Hence, our tactic to prove these obligations automatically uses several built-in decision tactics of HOL: TAUT_TAC, MESON_TAC, DECIDE_TAC, REDUCE_TAC and several tactics dealing with arithmetics. A detailed discussion of each of these tactics can be found in [24].

In our set of case studies, 37 of the 62 proof obligations concern domain checking, thus forming the largest category. The success rate of our tactic for this category is surprisingly high; all of the 37 proof obligations are proved automatically. Although, in general, the proofs belonging to this category may be relatively easy, they occur frequently and hence automation can save the user much time and effort spent on manual proof attempts.

5.2 Subtype checking

As mentioned in Section 2.1.2, the subtype proof obligations mainly result from the use of invariants and union types. In general, the manual proof that an invariant is satisfied is rather complex. The core of such a proof usually consists of a reasonable amount of applications of inference rules. But reaching this core requires working through a maze of definitions, simplifications and rewrites. Furthermore, the proof obligations themselves become rather long, making manual proofs even more complex, tedious and error-prone.

As a start, we tried to use our tactic for the domain checking obligations, but this was not very successful. Consequently, we have defined a new tactic based on the combination of two main approaches to find a proof: decision support and simplification support. They are described in the two paragraphs below.

Decision support.

Similar to our tactic for the domain checking proof obligations, part of our tactic for subtypes does regular logical reasoning. However, the simple logical inference rules of the previous tactic are no longer sufficient. They are extended by an additional set of inference rules, allowing for reasoning at a higher level of abstraction. The main concern about this extension is the number of inference rules added. Adding many rules will degrade the performance of the tactic dramatically. In the current tactic, we have found a small set of inference rules that leads to a large amount of proofs in our case studies.

Simplification support.

Simplification support is mainly based on rewriting tactics that allow faster and more powerful rewriting than decision tactics. Our approach primarily consists of using the so-called 'stateful' or 'implicit' simplification set, which basically involves the entire context of the proof obligation considered. This includes model definitions, type definitions, all loaded libraries and proof obligations that have already been proved. In practice, simplification tactics will not only rewrite the definitions in an obligation, but they will also rewrite the obligations to more basic forms.

The combination of decision support, which is relatively slow and uses only a small set of inference rules, and simplification support, which is fast but usually not able to complete the proof, results in a very powerful tactic. In our case studies, the subtype proof obligations occur 19 times out of the total of 62. By means of our tactic for subtypes, 14 out of these 19 are proved automatically. Since the tactic is very fast, there is still much room for improvement by adding theorems. However, finding the most suitable theorems to add is clearly the most complex task in the process.

5.3 Satisfiability and Termination

The last two categories of proof obligations, satisfiability and termination are the smallest in our case studies, corresponding to 6 out of the 62 proof obligations. In general, these proof obligations are either extremely hard to prove or rather trivial. In our case studies, 5 out of the 6 are proved by our tactic for domain checking obligations. The remaining proof obligation is a satisfiability problem which is hard to prove automatically. Not because of the length or complexity of the postcondition, but because the expression that could be used to satisfy this postcondition (i.e. the expression that could be used as function body) is far from trivial.

HOL automatically attempts to prove termination when defining a function. The current tactic of HOL can prove termination of several recursive functions. If this succeeds we can easily make use of the resulting theorem to prove our termination obligation.

6. CONCLUDING REMARKS

To ease the formal development of software by means of VDM, we have provided tool support to discharge a substantial part of the VDM proof obligations automatically using the HOL theorem prover. We have constructed several tactics in HOL to automatically discharge many of the proof obligations. Compared to earlier work [3, 12], our translation and HOL tactics use the HOL possibilities as much as possible, allowing for easier tactic construction and improved results. Our approach has been evaluated on four case studies for which 90% of the proof obligations have been proved automatically.

An important concern of the project was to limit the time required for individual proof attempts. Hence our tactics were developed such that they remained time efficient and the proof is performed fast, with a limit of ten seconds. In our case studies, most proofs take no longer than a second[2]. Running the tactics on all 62 proof obligations took no longer than two minutes. Hence, our approach allows for a quick check of consistency, where the user only has to focus on the unproved obligations.

Judging by these results and the additional knowledge gained using the case studies, we conclude that our approach is very useful in automating consistency proofs, thus preventing tedious and error-

[2]The machine used for testing was a AMD Athlon 2500+ notebook running Ubuntu Linux.

prone manual proofs. Furthermore, the HOL version of the model can also be used for easier manual proof of the proof obligations that could not be proved automatically. This facilitates the task of the modeler to ensure model consistency and to obtain executions that are free of run-time errors.

There are several topics for further research. To achieve mainstream usage, completion of the translation with the currently unsupported features is ongoing. Mutually recursive functions are being investigated and next support for the non-functional subset and VDM++ constructs will be addressed. On the proof-side, the tactics should be applied to a larger base of case studies. This will probably make them more widely applicable. Note that time efficiency of the current tactics is very good, leaving room for further additions and extensions. To improve the applicability and usability of the tool, it is relevant to investigate proof guidance at the VDM level (e.g., inserting proof hints in the VDM model) and the possibilities for feedback about the proof in the VDM language.

Acknowledgments

The authors wish to thank Nick Battle, John Fitzgerald, and Frits Vaandrager for their valuable comments and support when carrying out this work.

7. REFERENCES

[1] *Formals Aspects of Computing*, 20(1), 2007.

[2] S. Agerholm. Translating specifications in VDM-SL to PVS. In *Theorem Proving in Higher Order Logics (TPHOLs'96)*, LNCS 1125, pages 1–16. Springer-Verlag, 1996.

[3] S. Agerholm and J. Frost. An Isabelle-based theorem prover for VDM-SL. In *Theorem Proving in Higher Order Logics (TPHOLs'97)*, LNCS 1275, pages 1–16. Springer-Verlag, 1997.

[4] S. Agerholm and P. G. Larsen. Modeling and Validating SAFER in VDM-SL. In M. Holloway, editor, *Fourth NASA Langley Formal Methods Workshop*. NASA, September 1997. Available from http://atb-www.larc.nasa.gov/Lfm97/proceedings/.

[5] B. K. Aichernig and P. G. Larsen. A proof obligation generator for VDM-SL. In *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 338–357. Springer-Verlag, 1997.

[6] D. Andrews, P. Larsen, B. Hansen, H. Brunn, N. Plat, H. Toetenel, J. Dawes, G. Parkin, et al. Vienna Development Method – Specification Language – Part 1: Base Language. Technical Report 13817-1, ISO/IEC, 1996.

[7] Automated Reasoning Group, University of Cambridge, Computer Laboratory. *The HOL System Description*, 2007. http://www.cl.cam.ac.uk/research/hvg/HOL.

[8] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, LNCS 61. Springer, 1978.

[9] H. Bruun, F. Damm, and B. S. Hansen. An Approach to the Static Semantics of VDM-SL. In *VDM '91: Formal Software Development Methods*, pages 220–253. VDM Europe, Springer-Verlag, October 1991.

[10] CSK Group. *VDMTools*, 2007. http://www.csk.com/support_e/vdm.

[11] F. Damm, H. Bruun, and B. S. Hansen. On Type Checking in VDM and Related Consistency Issues. In *VDM '91: Formal Software Development Methods*, pages 45–62. VDM Europe, Springer-Verlag, October 1991.

[12] L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, G. Robinson, M. J. C. Gordon, and T. F. Melham. The prosper toolkit. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '00)*, LNCS 1785, pages 78–92. Springer-Verlag, 2000.

[13] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[14] J. Fitzgerald, P. G. Larsen, and S. Sahara. VDMTools: advances in support for formal modeling in VDM. *Sigplan Notices*, 43(2):3–11, 2008.

[15] J. S. Fitzgerald. The Typed Logic of Partial Functions and the Vienna Development Method. In D. Bjørner and M. C. Henson, editors, *Logics of Specification Languages*, EATCS Monographs in Theoretical Computer Science, pages 427–461. Springer, 2007.

[16] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna development method. In B. W. Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.

[17] T. Kurita, M. Chiba, and Y. Nakatsugawa. Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods*, LNCS 5014, pages 425–429. Springer-Verlag, 2008.

[18] T. Kurita and Y. Nakatsugawa. The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3), 2009.

[19] P. G. Larsen and J. S. Fitzgerald. Recent industrial applications of VDM in Japan. In J. P. B. P. Boca and P. G. Larsen, editors, *Proc. BCS-FACS Workshop on Formal Methods in Industry, Electronic Workshops in Computing*. The British Computer Society, 2008.

[20] P. G. Larsen and W. Pawłowski. The Formal Semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602, 1995.

[21] H. D. Macedo, P. G. Larsen, and J. S. Fitzgerald. Incremental development of a distributed real-time model of a cardiac pacing system using vdm. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods*, LNCS 5014, pages 181–197. Springer-Verlag, 2008.

[22] S. Maharaj and J. Bicarregui. On the verification of VDM specification and refinement with PVS. In *Conference on Automated Software Engineering (ASE '97)*, pages 280–289. IEEE Computer Society, 1997.

[23] NASA. Formal methods, specification and verification guidebook for software and computer systems - a practitioner's companion. Draft 2$\dot{0}$, Washington, DC 20546, USA, November 1996.

[24] M. Norrish, K. Slind, et al. *The HOL System reference*. Automated Reasoning Group, University of Cambridge, Computer Laboratory, 2007. http://hol.sourceforge.net/documentation.html.

[25] Open-source Tools for Formal Modeling. *The Overture Initiative*, 2007. http://www.overture.org.

[26] S. Stepney, D. Cooper, and J. Woodcock. *An electronic purse: specification, refinement, and proof*. Technical monograph PRG-126. Oxford University Computing Laboratory, July 2000.

[27] VDM-Team. The VDM++ Language Manual. Technical report, CSK, 2007. http://www.vdmtools.jp/uploads/manuals/langmanpp_a4E.pdf.