Sander Daniël Vermolen

Automatically Discharging VDM Proof Obligations using HOL

August, 2007

Radboud University Nijmegen
Computing Science Department

Supervisors

Dr. J.J.M. Hooman
Prof. dr. F.W. Vaandrager
Dr. P.G. Larsen

Contents

Abstract

Statically undecidable inconsistencies in VDM++ models can result in runtime errors when executing the model. The absence of these inconsistencies can be verified by proving the corresponding proof obligation to be valid. This thesis comprises two main steps. The first is a translation of functional VDM++ models to semantically equivalent HOL models. The second is an automated proof of the obligations that are generated from a VDM++ model. The combination of these two steps can ensure consistency of the model and thereby stability of the execution.

Dutch abstract

Statisch onbeslisbare inconsitenties in VDM++ modellen kunnen executie fouten opleveren zodra het model wordt uitgevoerd. De afwezigheid van deze inconsistenties kan worden geverifieerd door de bijbehorende bewijs verplichtingen correct te bewijzen. Deze scriptie bestaat uit twee stappen. De eerste stap is een vertaling van functionele VDM++ modellen naar semantisch equivalente HOL modellen. De tweede is een geautomatiseerd bewijs van de verplichtingen die uit een VDM++ model zijn gegenereerd. De combinatie van deze twee stappen verzekert de consistentie van het model en daarmee de stabiliteit van de uitvoering.

Acknowledgements

I would like to thank Jozef Hooman for his excellent supervision of the project. His broad view and eye for detail have greatly aided me during my thesis.

Many thanks go to Peter Gorm Larsen, who has also enthusiastically supervised a large part of the project. Being open, friendly and patient, his help was much more widely applicable than just to the thesis. Thank you for all valuable advice and useful comments.

Thanks go also to John Fitzgerald, who enabled two very useful visits of Newcastle University, which both greatly helped in the development of the tactics.

I would also like to thank Frits Vaandrager for being the second supervisor and Marcel Verhoef for providing the opportunity to do this project in the first place.

Chapter 1

Introduction

Although it usually goes unnoticed, software is present nearly anywhere we go. It controls anything from the movement of our car, the balance of our bank account and the remote control of our television, to nuclear power plants, space ships and airplanes. Almost as present as the software itself, are the mistakes that were made during development. The malfunctioning of a remote control may be annoying, but the malfunctioning of the control of a nuclear plant may be devastating.

It is therefore no wonder that software correctness is one of the most important topics in software engineering. Although full confidence that software is entirely correct is usually considered to be unreachable, increasing confidence in software is the focus of much research today. Many techniques are available to achieve this goal, among which is the use of formal methods.

1.1 Formal methods

We can distinguish many kinds of software engineering methods. One way to distinguish between them is based upon their 'formality'. Formality can be said to be loosely tied to the degree of mathematical rigor that is applied during analysis and design [30]. Formal methods cover the far end of the formality scale. They can be described as the applied mathematics of computer system engineering [33].

Formal methods are controversial. On one side, they are considered to be powerful and able to revolutionize software development, while on the other side they are considered impossibly difficult [22]. It is therefore especially the (apparent) difficulty that prevents a wide acceptance of formal methods into all (or at least most) branches of software development. The focus of this thesis will be on reducing this difficulty in one specific formal method, by automating as many complex tasks as possible and leaving only the usage of the results of the method up to the user.

One kind of formal method is a formal specification method. Formal specification has its focus on specification using a mathematically based language. The goal is to systematically describe the (desired) behavior of a system. Expressiveness of formal specification is usually not restricted to software, they

may for example also describe the behavior of a physical system.

Many formal specification methods are available and used in software development. Some of these are [16]:

- Abstract State Machines (ASM)
- B-Method
- CommUnity
- Petri Nets
- Uppaal
- Vienna Development Method (VDM)
- Overture Modeling Language (OML)

The last two provide a large part of the context of this thesis.

1.2 VDM

The Vienna Development Method (VDM) is a collection of techniques for the modeling, specification and design of computer-based systems [20]. Throughout the years, several specification languages have supported the VDM principles. These include VDL, VDM-SL, VDM++ and OML. VDM++, which is an Object Oriented extension of VDM-SL, will be used in this project.

VDM is supported by a closed-source set of support tools called VDMTools. This has support for parsing, type checking, interpretation, debugging, Java code generation and many more activities. An open-source initiative called Overture [8] is currently developing a somewhat more sophisticated set of support tools. This thesis is part of the Overture project. As the project just started, the only part of Overture that was finished and could be used for this thesis was a parser. Other VDM tools that were required, were used from the VDMTools set.

VDM++ itself is text-based and has at first sight a lot in common with many other Object Oriented languages such as Java or C++. In contrast to these, VDM++ is focused on modeling instead of programming and supports invariants, pre- and postconditions and even non-executable models. Furthermore, VDM++ has (as all other VDM languages) a formally specified semantics. We will see many examples and more extensive explanations of these later in the thesis.

A more extensive description of VDM can be found in Section 2.1.

1.3 Consistency of VDM models

Just as any piece of software, VDM models can contain errors. We can distinguish two kinds of errors: those that can be found purely by analysis of the model and those that can only be found when more information than just the model is given.

The last type is in practice hard to detect automatically, since the extra information required to find them is usually not available to a computer. For example, suppose someone wants to develop a function that always returns the number two, but instead accidentally writes a function that returns the number one. To detect this, the intentions of the developer need to be uncovered, which is rather difficult for a computer.

There is also a large body of common and uncommon errors that can be detected without any additional information. These errors are called inconsistencies of the model and can typically trigger 'runtime errors' when executing the model. Examples of inconsistencies are type errors.

When looking at inconsistencies, we can distinguish several kinds of VDM specifications (models). They are represented graphically in Figure 1.1. All

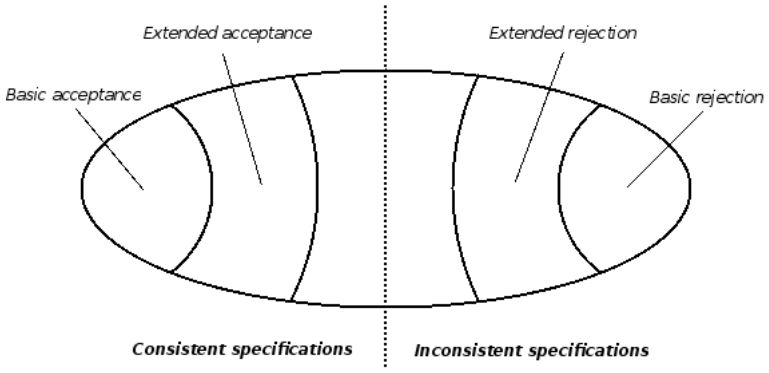

Figure 1.1: Types of specifications

consistent specifications are represented by the left half of the figure, all inconsistent by the right. When eliminating as many inconsistencies as possible, we would want to reject as many inconsistent specifications as possible, but to help the developer, we would want to accept as many consistent models as possible. As we will see later, rejecting all inconsistent and/or accepting all consistent specifications is unfortunately not possible.

Within each of the two categories in the figure (consistent and inconsistent), we can distinguish three more groups:

1. *Basic* These models can statically be determined to be accepted or rejected. In the case of VDM, this means a type checker (of VDMTools, or Overture) is able to determine its consistency or inconsistency.

2. *Extended* A static test cannot give a conclusion on these models. They are statically undecidable. They may show behavior when executed, that cannot be determined easily without executing them. In addition to the static checker, a more sophisticated, dynamic check is therefore required. This check will try to prove consistency by reasoning about the execution of the model.

3. *Unprovable* The last category covers the middle of the figure. It holds all specifications that cannot be determined to be either consistent or

inconsistent in one of the two ways above. Not by a static check and not by a dynamic check.

Most of the models in the second and third category are caused by the static undecidability of many VDM types. For example, consider the following piece of code:

```
x + 1
```

When `x` is an integer (just as `1`), this piece of code would be type correct. However, when we define `x` to be of a union type of strings and integers (meaning that `x` is of type string *or* of type integer), then the expression is only type correct if when executing the expression, `x` is actually an integer. To determine that `x` is indeed an integer for any execution of the model, one needs the context of the expression. Of course there are contexts which make the consistency obvious. Take for example:

```
if   x is of type integer
then x + 1
```

However, in practice, these contexts are much more complex and their consistency cannot be determined easily anymore.

In order to determine the consistency of a specification one needs to know 'where' to search for inconsistencies. If all locations where inconsistencies might occur prove to be consistent, we can say for certain that the model is consistent. Fortunately it is relatively easy to locate potential inconsistencies in a model. They are always caused by specific code constructions in the model. All statically decidable inconsistencies will be verified by the type checker. All remaining inconsistencies (the ones that are statically undecidable) will be located by the integrity checker of VDMTools. This tool checks for 51 types of potential inconsistencies in the model. Whenever a potential inconsistency is found, the integrity checker can generate a predicate. Whenever the predicate is valid (always true), there is no inconsistency at this specific location, when it is not valid, there will be an inconsistency. These predicates are called *proof obligations*. Together with the integrity checker, they reduce the problem of finding statically undecidable inconsistencies to proving a set of predicates.

1.4 Theorem provers

Since mathematical proof is frequently a complex and difficult task, there are various pieces of software to help in this process. A theorem prover is one of them. When using a theorem prover, the proof usually starts by entering a predicate that needs to be proved to be valid (in a language suitable to the prover). In the end, if a proof is found, the proved predicate will be called a 'theorem'.

In general, two ways of proving theorems can be distinguished: interactive and automatic proof. Using the first method, the user will have to give a detailed description of how the theorem prover is supposed to do the proof. The prover itself will do the administration (such as looking up requested theories

and keeping track of sub goals). In the second method, the prover will try to prove a theorem itself, but depending on the specific automation, limited user input may be required.

Although an automated proof may seem very useful, in practice its applicability is relatively limited. The automated execution is just an extensive search, hoping to find the desired proof. This means that the prover has to be told in advance exactly which proof attempts it should try. A description of the attempts it should go for and in what way (such as what order) is called a 'tactic'.

If a large number of proof attempts are selected in a tactic, the search for a proof will in general take longer, but for more theorems a proof might be found. If fewer proof attempts are selected, the search is usually more computationally efficient, but less effective in finding a proof. The trade off to get a good tactic is usually between the computational and functional efficiency: to get the computationally fastest tactic, that is able to proof all or most of the theorems one wants to prove.

Higher Order Logic (HOL) is a theorem prover that will be used in this project. More information on HOL can be found in Section 2.2.

1.5 Problem description

As we have seen in the example in the previous section, usage of different types is one of the sources of inconsistencies in a model. In addition to this, there are more language components that might provide statically undecidable issues. These include for example termination and domain satisfaction of partial functions (although the last can be considered to be a type issue too).

Since consistency of a model is a useful and in many situations even a critical property of a model, the proof obligations provided by the integrity examiner give us a useful aid when developing systems. Unfortunately proving the obligations to be valid manually is a tedious and time consuming task. Especially when dealing with larger and more complex models, in which the number of obligations and their complexity can increase rapidly.

1.5.1 Goals

The goal of the work described in this thesis is to reduce the amount and complexity of manual proof by automating, as far as possible, the discharging of proof obligations generated by the integrity examiner of VDMTools. The theorem prover HOL will be used to supply the automated proof. As this work is done within the context of the Overture project, the integrity examiner of VDMTools will most likely eventually be replaced by an equivalent within the Overture project, but at the time of writing, this does not exist yet.

Two main sub goals can be distinguished:

1. *Development of a VDM++ to HOL translator*
   Since HOL will be used to prove the obligations, they have to be translated

to a language that HOL can read. Crucial to the success of the translation and especially the success of the proof to come, is the semantical equivalence between the two models.

2. *Proof of the proof obligations*
   Using the result of the translation as input into HOL, the next goal is to prove as many of the proof obligations of the model as possible. Tactics are the main tools to reach this goal. They are the means to guide the search for a proof and hopefully find one.

Achieving exactly this kind of automated proof has already been attempted in the PROSPER project [4], which was able to discharge 90% of the obligations in one of the larger case studies used. The tool set developed in PROSPER used HOL98 (an older version of HOL), to prove obligations of the language VDM-SL. Despite its promising results, the tool set was never further developed and the sources of the conversion have been lost. Furthermore, there were parts of the VDM-SL language, which were not or were only partially implemented.

Considering the increased power of theorem provers (HOL98 has been incremented to a new version, called HOL4), there is also good reason to believe that a higher level of automated proof can be reached. This project might not be able to use this yet to its full extent, but is using a large part of it and has developed a basis to support it further. Some ideas of PROSPER, yet limited in number, were used along the way.

1.5.2 Main challenges

There are many challenges on the way to achieving the two sub goals. The major challenges in translation are:

- *Semantical equivalence:* This equivalence is the key concept of the translation. Without it, the derived proof has lost its meaning. Since the equivalence is hard to determine formally (since we are dealing with two formalisms), the thesis will try to increase confidence in the translation by showing equivalence informally.

- *Difference in language constructions and functions:* Some of the VDM language constructions (such as implicit functions) and functions (such as unequal) are not defined, or defined differently in HOL. This requires appropriate rewriting. We will see throughout this chapter that this is not always as easy at it may seem at first sight. Especially when not only the function definition itself is different, but also the principle of the definition (as is the case for example with maps and records).

- *Difference in types:* We will see that most types are relatively easy to translate, as they have an equivalent or a near-equivalent in HOL. The way the types are defined is however not straightforward.

- *Patterns:* VDM has patterns in all kinds of constructs (cases expression, all types of bindings, function definitions, etc.). HOL on the contrary has very few. Since the use of patterns is so common in a VDM model, the translation of these will be required and far from trivial.

- *Partial to Total:* All partial functions in VDM need to get some kind of total representation in HOL, as HOL does not allow partial functions.
- *subtyping and Type unions:* VDM supports subtyping and union types, while HOL does not directly support subtyping and non-disjoint union types. Also, type equivalence in VDM is structural whereas HOL uses a name-based equivalence.
- *Object Oriented:* VDM is object oriented, while HOL is not.

Some major challenges in the proof are:

- *Finding the right effectivity-computability trade-off:* Since a proof attempt is no more than a search for a proof, a good trade-off between search time and number of proved theorems has to be found.
- *Improving tactic speed without making concessions to its performance:* A user will not only want his proof obligations to be proved, he also wants them to be proved as fast as possible. This can be done by doing concessions on the number of proof attempts (the previous challenge) but also by improving the tactics' speed without losing attempts.
- *Improving the tactics' (and HOL's) applicability by extending the theorem base:* As most theorem provers, HOL has a large base of already proved theorems, that can be used right away in other proofs. Since HOL was not specifically designed to work for VDM proofs, extensions of these libraries may provide additional applicability and possibly even additional speed.

1.6 Approach

The goals stated in Section 1.5.2 can be reached in various ways. This section will look at how these were achieved in in this project and why these choices were made. First we will look at the development methodology of this project. After that, the focus will be on the different aspects of the proof automation itself.

1.6.1 Development methodology

As there are many ways to solve the translation and many ways to implement the proof, additional aid is required to achieve the goals in a right way. This aid will come in this project from case studies. The models used in these case studies are intended to generate a typical or "average" selection of proof obligations. Yet at the same time they must provide enough challenge to make progress in development of the tactics and translation. In total, about 10 case studies have been used. Most of them have originally also been used in the PROSPER project.

The case studies have been used in an iterative approach. For every case study, the translation was improved to support all language constructs used in the case's model and the tactic was improved to be able to find as many of the case's proofs as possible.

1.6.2 Proof automation architecture

The main part of the proof automation consists of three steps:

- *a preparation step* in which the input VDM model is converted to a format suitable for processing and the proof obligations that need to be proved are generated;
- *a translation step* in which the output of the previous step is converted to a format providing the basis for the final HOL model and
- *a proof step* in which the actual HOL model is generated and an attempt to prove the obligations is made.

The following sections will focus on each of them individually.

1.6.2.1 Preparation

Although a concrete VDM model is very suitable for human processing, in automated processes a more explicitly structured and usually somewhat more redundant representation is preferred. A tree structure is easier to process than the linear text in the model and components of the model that have been assigned their specific function are easier to process than components of which the function must be deduced from the context. Even though all this information is contained within the concrete model, a conversion of representation to a format that makes it more explicit is useful in later processing. We will refer to the result of this conversion as the VDM Abstract Syntax Tree (VDM AST) or when an entire VDM model was used as input to the conversion, the abstract VDM model. The first task of the preparation step is this conversion of the concrete syntax VDM model to its VDM AST.

The second task in preparation is the generation of the proof obligations of the model. They are usually extracted from the AST generated in the previous task and will also usually be provided as VDM AST's.

Both these tasks are performed by existing software. In VDMTools this would be the parser in combination with the integrity examiner. In Overture no equivalent to the integrity examiner is available yet at the time of this writing, so only the Overture parser can be used. But eventually, the open-source software from Overture should be the main candidate to execute these tasks. Figure 1.2 gives us a schematic representation of the preparation phase. The larger blocks indicate intermediate instances (e.g. the models or proof obligations), the smaller, dotted blocks indicate processing of these to new intermediate formats (or eventually to the resulting proofs. The gray color indicates that this part of the processing falls outside the scope of this project's code and will be a prerequisite.

1.6.2.2 Translation

Neither the VDM AST nor the concrete syntax VDM model can be used directly as input to the HOL engine. HOL uses its own language to express models and to control the engine. A translation from VDM to HOL is thus required. Using the abstract VDM model and the proof obligations from the preparation phase, the first task of this step involves this translation. The translation is shown in

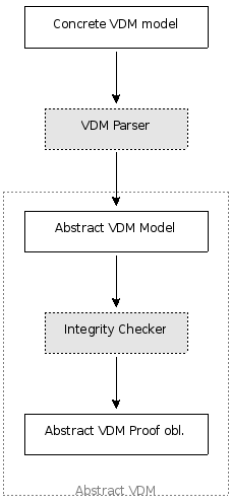

Figure 1.2: First two phases of document processing, that will provide the input to the project

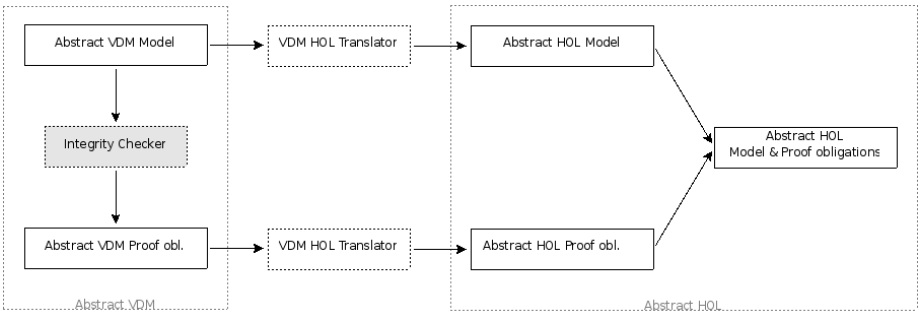

Figure 1.3: Translation of an abstract VDM model to an abstract HOL model

Figure 1.3. The translator that is defined and implemented in this project (see Chapter 3), uses the abstract VDM model as input and produces an abstract HOL model as output. As also the proof obligations are available as VDM abstract syntax trees from the previous step, the same translation can be used to convert the obligations. The two translator boxes in the figure therefore use the same code.

The entire translation will be specified as a VDM model itself. This model can be interpreted to execute the translation. When using it in the Overture project, most likely Java code will be generated from the translator model, which is then compiled and interpreted to perform the translation.

Once translated, we enter the 'HOL domain'. From now on, all processing will be in HOL terms (either concrete or abstract). The second task of the translation will be a merge of the proof obligations and the model. Apart from just joining the two code bases, this merge also consists of type rewritings of the proof obligations using the document's definitions. This step will be explained in detail when these type issues arise in Chapter 3.

The integrity checker in the figure does not only result in the proof obligations themselves, although this is suggested by the figure. We get much more information about the obligations (such as the type and location), that have been left out of the diagram for simplicity. However, this information will be kept in the translation, allowing the next step to make use of it.

1.6.2.3 Proof

The final and seemingly most important part of the process is the proof (shown in Figure 1.4). Using the abstract HOL model and its proof obligations resulting

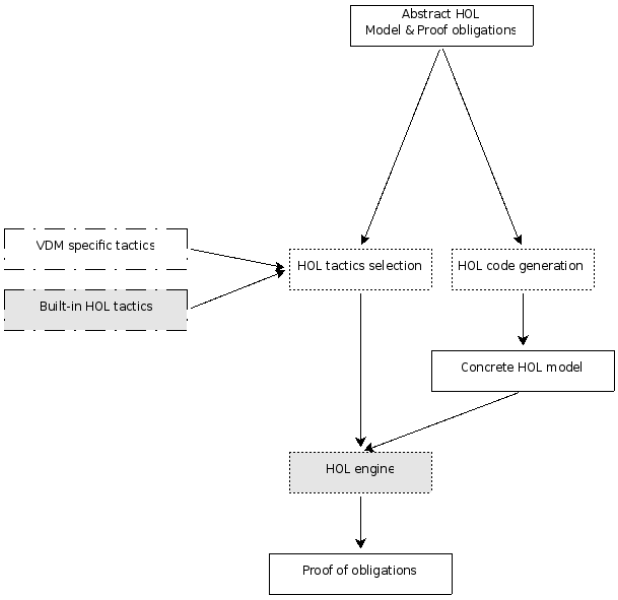

Figure 1.4: Proof of a document that results from the translation

from the previous step, there are two tasks left to be executed before actually using the HOL engine:

- *HOL code generation* This involves serializing the abstract HOL syntax tree to a concrete HOL model, such that it can be used as input to a HOL session. Note that this task might be considered to be part of the translation. The reason that it is not included there is that the abstract HOL model is a prerequisite of this task, not the concrete one. This abstract model might be altered depending for example on the available HOL tactics. Not to change the semantics of the model, but to influence the way HOL handles it. We will see an example of this later when dealing with quantifiers: There are several ways of expressing the type of a variable bound by a quantifier. A choice of which way to use might depend on the tactic that is being used or the type of proof that is to be performed.

- *HOL tactics selection* A good tactic is crucial for being able to prove the obligation and being able to do it in a reasonable amount of time. We can choose from a base of ready-made HOL tactics, that are powerful and

generic (called 'Built-in HOL tactics' in the figure). Additionally we can use custom-made tactics, that we can design specifically for the purpose of a VDM proof (called 'VDM specific tactics'). Both these tactic bases are separated from the other steps and intermediate formats, because they are static and independent from the model[1].
Considering that the number of tactics that can be deduced from these two bases is practically infinite, a good selection and ordering of the tactics is required to save time in the process of proving the obligations. Simply trying a large quantity of them would be time consuming and a risk that the operation is aborted prematurely by the user, possibly loosing results gained along the way. Thankfully, the additional information on the proof obligations, mentioned earlier, will come in handy at this point.

Once we have our concrete HOL model, and selected tactics, we can start the proof attempts themselves. This starts by loading all required libraries into HOL, followed by loading the model and selected tactics. Next, the attempts to prove each of the individual proof obligations will finish the work done in the previous steps.

1.7 Structure of this thesis

The thesis is structured as follows. The next chapter (2) will present background information by looking more deeply into the knowledge and earlier development concerning the topics involved in this thesis. Chapter 3 will focus primarily on the translation of a VDM++ model to a HOL specification, followed by a discussion of the second step of the project, namely the proof of the obligations in Chapter 4. Chapter 5 will give us more insight into the case studies that were used. A discussion of related work can be found in Chapter 6. The thesis will conclude with Chapter 7, in which the conclusions and options for further research are explained.

[1] We will see in chapter 4, that this is not entirely true. Although the way the tactics work is static, they are not static themselves. For simplicity, this is left out of the model used here.

Chapter 2

Background information

This chapter will mainly focus on the two languages used throughout the project: VDM++ and HOL. Section 2.1 will first give a brief discussion of part of the design of VDM++ that is crucial throughout the rest of this document. Then it will explain the origin and development of VDM. Finally, it will focus on two tool sets that support VDM++. Section 2.2 will give a description of part of the design of HOL and its development. The last subsection is devoted to the meta language used to control HOL, namely ML.

2.1 Vienna Development Method

The Vienna Development Method (VDM) is a collection of techniques for the modeling, specification and design of computer-based systems [20]. Because of its formally defined semantics, it is suitable for formal proofs. This section will first look at the general design of VDM, then at its development throughout the years. Of the three remaining sections, the first two will focus on two tool sets for VDM and the last will look at the VDM language itself.

2.1.1 VDM Design

Although the primary focus of VDM is on software, it is designed to be a modeling language applicable to a wide range of systems. The language has a formally defined semantics and allows for formal reasoning [15].

The languages of VDM provide constructs to write executable as well as non-executable models. Some of the tools (such as an interpreter) only focus on executable models, but most of the tools support both types. Since we will look at consistency of models in this thesis, both executable and non-executable models will come into focus.

There are two main ways of specifying a VDM model: functional and operational. The first only provides programming by 'mathematical' functions. It avoids states and mutable data. The second provides a range of additional constructs, that *can* deal with state. Typical non-functional language constructs are loops and assignments. Any functional model is also an operational one, the operational part simply allows for more constructs. For reasons that will be discussed later, this thesis will only look at functional models.

Several languages have been used within the VDM methodology throughout the years. In this thesis, only the languages VDM-SL, VDM++ and OML are relevant. VDM-SL [23] is the standardized language supporting VDM, VDM++ [?] is an object oriented extension (and slight adaption) of this and OML is the Overture Modeling Language, which is a variant of VDM++. The main focus of this thesis will be on VDM++, which is, in the constructs used here, very similar to OML. VDM-SL may sometimes be mentioned when discussing contexts.

2.1.2 VDM Development

VDM has its origins in the 1970's at the IBM Research Lab in Vienna. It was originally developed to provide a formal semantics of the PL/I programming language as part of compiler correctness arguments. It has since been applied to a much wider range of application domains. A good discussion of the technical decisions underpinning VDM can be found in [24].

2.1.3 VDMTools

VDMTools [6] is a closed-source tool set supporting VDM-SL and VDM++. VDMTools was originally developed by the Danish company IFAD A/S (Instituttet for Anvendt Datateknik). The intellectual property was subsequently acquired by CSK Systems Corporation, which now maintains and further develops the tool set.

VDMTools mainly consists of a parser, syntax checker, integrity examiner, an interpreter and debugger, a UML link and conversions of VDM models to various programming languages (such as Java). VDMTools is currently the main set of VDM-SL supporting tools used in industry. When referring to the VDMTools set, in this text, the focus will mainly be on the parser, syntax checker, or the static semantics checker. Although the other tools have been used in the course of the project, they are not directly relevant to the results.

2.1.4 Overture

Overture is a 'community-based' open source project to develop a modern, industrial-strength alternative to VDMTools. It uses an extension of VDM++ called Overture Modeling Language (OML) and is supposed to be integrated into Eclipse as a plugin eventually. This thesis is part of the Overture project and intended to provide it with additional functionality and knowledge to further develop the Overture tools.

2.1.5 Functional subset of VDM++

Since some basic knowledge of the VDM++ language is useful throughout the rest of this document, this section will give a brief description of part of the functional subset of VDM++. A much more extensive and complete description can be found in [19].

2.1.5.1 Types

VDM++ provides a rich typing system. All expressions in the language are typed (implicitly or explicitly) and it allows for a wide variety of type definitions

and usages. The language includes a small set of basic types, such as the natural numbers: `nat`, the real numbers: `real` and the booleans: `bool`. Added to these is a class of types which is somewhat less common: the quote type. Quote types are types that only contain a single value. Typically, the name of the type is also the name of the value. To distinguish the value from the type, the value is written between < and >. For example, the only value of the quote type `Red` is:

```
<Red>
```

In addition to the types above, VDM++ has several type constructors: Given one or more existing types and a type constructor, a new type can result. The most common are:

union Given two types, this will construct a type which is a union of the two. The VDM++ representation of union is the vertical bar. We could for example define a type that is either a boolean or a natural number by:

```
BoolOrNat = bool | nat;
```

Using the quote type above, we can represent what is in many languages known as an enumeration type:

```
Color = <Red> | <Orange> | <Green>;
```

The only operations available on values in unions are equality and inequality (= and <>).

set Only one type is required to construct a set type: the type of the elements in the set. Using the type Color defined above, we can define the number of possible colors of a traffic light to be a set of colors:

```
TrafficLightVariety = set of Color;
```

For a pedestrian light (usually only red and green), the set would be written as:

```
{<Red>, <Green>}
```

There are many operations available on sets. These include membership test, union, intersection, subset and proper subset. Sets in VDM are always finite.

map Given a source and target type, the map constructor creates a type that holds finite mappings from the source type to the target type. The pattern in which a traffic light changes color can be modeled using such a mapping:

```
Pattern = map Color to Color;
```

To instantiate a mapping, we would write for a traffic light using the sequence Red, Green, Orange, Red (A Dutch one):

```
{<Red> |-> <Green>, <Green> |-> <Orange>, <Orange> |-> <Red>}
```

The main usage of a mapping is applying it to a value in its domain, which can be written down as an ordinary function application. But there are many more operations available on maps, such as acquiring domain or range, or merging and restricting maps.

product A product type can use any number of types to construct it, depending on the number of fields in the product instances. The `*` is the type operator to construct a product type. A location modeled by two coordinates would be:

```
Location = nat * nat;
```

The instantiation of a value of this type looks like:

```
mk_(2, 3)
```

The `mk_` is mandatory and used for any product. The number of fields can vary depending on the definition.

function Where map is a finite mapping, the function type can (but does not necessarily have to) hold an infinite mapping. A function type can be constructed using an arrow. For example the type of a summation function can be written as:

```
sumType = nat * nat -> nat;
```

Making instances of this function type is by defining functions. We will see examples of this later when discussing function definitions.

record Records are mappings of some fixed field names to a value assigned to that field for a particular record. Basically records are products, with an extra option to give each of the fields a name. Using the previous definitions, we can define a traffic light to be a record:

```
TrafficLight :: currentColor : Color
                availableColors : TrafficLightVariety
                pattern : map Color to Color            ;
```

Unlike many other languages, the order in which the record fields are defined is of importance. It is for example required when defining a new record. A regular Dutch traffic light set to Red would be:

```
mk_TrafficLight(
        <Red>,
        {<Red>, <Orange>, <Green>},
        {<Red> |-> <Green>, <Green> |-> <Orange>,
         <Orange> |-> <Red>}
)
```

To select a field out of a record, an infix dot operator is used (.). On the left hand side of the dot, the record is specified, after the dot one of the field names defined in the type.

In VDM++ it is also possible to state invariants on these types. For example, in the traffic light defined above, the current color and all colors used in the pattern should be in the set of available colors: a traffic light cannot turn to a color it does not have. In a VDM++ model, we can state this explicitly by using the `inv` keyword in the type definition:

```
TrafficLight :: currentColor : Color
                availableColors : TrafficLightVariety
                pattern : map Color to Color
inv currentColor in set availableColors and
    dom pattern  subset availableColors and
    rng pattern  subset availableColors                ;
```

Here `dom` and `rng` select the domain and range of a map and `subset` is a regular (non-proper) subset operator. The element-of operator is represented by the name `in set`.

2.1.5.2 Functionality

Most of the VDM++ language used to specify functionality is relatively similar to programming languages (especially functional programming languages when dealing with the functional subset). In the discussion below, I will only discuss parts that are not straightforward or less common.

Set comprehension In order to express selection and processing of several elements out of a set more easily, VDM++ supports set comprehension. A set comprehension consists of three parts: construction of the new elements, bindings to a set and a test to select only specific elements. We could for example convert the map used in the traffic light pattern above to a set of tuples. But leave out the mapping of the current color. This can be written as:

```
{mk_(color, trafficLight.pattern(color))  |
    color in set dom trafficLight.pattern &
    color <> trafficLight.currentColor      }
```

Each line corresponds to one part of the set comprehension. The first line creates the tuples. The second specifies the source of the colors and the third makes sure that the current color is left out. The variable `trafficLight` is assumed to be of type `TrafficLight`.

Related forms of comprehensions exist that are applicable to sequences and sets.

Let VDM++ contains two 'let' expressions. One is the let-in that can be found in many other languages. It can be used to assign values to local variables:

```
let
  x = 1,
  y = 2
```

```
in
  x + y
```

This will result in 3.

The other 'let' expression is the let-be-such-that. This will select a single, arbitrary value from a type or set, that satisfies a given predicate. This value will then be assigned to a local variable. For example:

```
let
  x in set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
be st
  x < 7 and x > 3
in
  x * x
```

Which will result in 16, 25 *or* 36.

Functions There are two types of functions: implicit and explicit. Explicit functions consist of a signature of the function, followed by an explicit specification of how to evaluate it. If desired, this may be followed by a pre and/or postcondition. The definition of an implicit function also starts with a signature, but does not have an explicit specification how to evaluate it. The postcondition on the contrary is mandatory. The precondition is still optional.

An example of an explicit function to set the color of a traffic light to a desired color is:

```
changeColor: TrafficLight * Color -> TrafficLight
changeColor (light, newColor) ==
  mk_TrafficLight(newColor, light.availableColors, light.pattern)
pre
  newColor in set light.availableColors
post
  RESULT.currentColor = newColor                                    ;
```

The first line is the signature of the function, containing the type of the function and its name. The pre and postcondition are indicated by the keywords `pre` and `post`. In this example, the precondition is required to make sure that the invariant stated earlier in the type definition of the traffic light holds (the color must be available on the traffic light). The postcondition is not required. The fact that it holds can easily be deduced from the function body. This might not be that obvious when dealing with more complex functions. The identifier `RESULT` used in the postcondition is a reserved identifier that will always stand for the result of the function.

An example of an implicit function definition of the same function is:

```
changeColor (l : TrafficLight, newColor : Color) r : TrafficLight
pre
  newColor in set light.availableColors
post
  r.currentColor    = newColor and
```

```
  r.availableColors = l.availableColors and
  r.pattern         = l.pattern                                    ;
```

The signature of the function has changed, now incorporating both the variable names, their types and a name of the result. The precondition remains unchanged. The postcondition now uses the defined result name (instead of the identifier `RESULT`) and is completed to exactly specify what the result should be.

Although implicit functions cannot be executed like explicit functions can, reasoning about them is still an option. They will therefore be included in the translation discussed later on.

Patterns In the preceding description of the VDM++ language, we have used identifiers to indicate a variable. Instead of these, in many locations, also patterns may be used, including variables in the let expression, parameter lists of functions and bindings in set comprehensions, but there are many more. In this document, only a limited number of patterns will be discussed. The most important ones that are relevant here are:

Identifier This is the use of an ordinary identifier. It is no different from the situations discussed above.

Don't-care When this type of pattern is used, it indicates that the value, whatever it would have been assigned to or compared with is ignored. The value is (or should be) irrelevant to the specific part of the model.

Record A record pattern can split up a record value into its fields directly without having to use the dot operator.

To clarify what the patterns can be used for, this is an example of their usage in a let expression:

```
let
  x = 1,
  mk_TrafficLight(cur, ava, pat) = trafficLightVariable
  mk_TrafficLight(cur2, -, pat2) = trafficLightVariable2
in
  x
```

The first line after the keyword let, holds an identifier pattern of identifier `x`. The second is an example of a record pattern, filled with identifier patterns. The result is, that `cur` will have the value `trafficLightVariable.currentColor`, `ava` will have the value `trafficLightVariable.availableColors` and `pat` will have `trafficLightVariable.pattern`. The third line does the same as the second to `cur2` and `pat2`. The value `trafficLightVariable2.availableColors` will be ignored.

Patterns can be used in the same way in most other locations. When their usage is different in this document, this will be explained first.

Classes Finally, it is possible to construct classes in VDM++. The definition usually looks like:

```
class ClassName

instance variables
  ...
types
  ...
functions
  ...

end ClassName
```

Instance variables, types and functions are keywords. Each of them is followed by their respective contents.

We could have defined the traffic light as a class instead of a record. Using the same definitions as above, it would look like:

```
class TrafficLight

instance variables
  currentColor    : Color;
  availableColors : TrafficLightVariety;
  pattern         : map Color to Color;

  inv currentColor in set availableColors and
      dom pattern  subset availableColors and
      rng pattern  subset availableColors     ;
types
  Color = <Red> | <Orange> | <Green>;
  TrafficLightVariety = Color set;

functions
  ...

end TrafficLight
```

Usage of classes is very similar to Java or C++. Their members can be called using the 'dot' operator and their equality is based on reference. It is also possible to assign access restrictions to the various members, but this is not relevant to this project and will therefore not be discussed.

2.2 HOL

HOL is an automated proof system for higher order logic. It was originally developed by the Automated Reasoning Group (University of Cambridge) and was originally intended for hardware verification. Currently it is a widely applicable programming environment in which theorems can be proved and proof tools implemented. The latest version, called HOL 4, is open source with a BSD-style license that allows free use in commercial products.

2.2.1 HOL Design

The HOL system is designed for interactive as well as automated theorem proving. It combines higher order logic (hence HOL) with typed lambda calculus [17].

HOL consists of two layers: a meta-level layer and an object-level layer. The HOL engine itself operates at an object-level. Controlling the HOL engine can be done through the meta-level, which is in the case of HOL implemented by a language called ML (short for Meta Language). There are two main types used by HOL worth distinguishing:

term An instance of this type always denotes a expression (e.g. $\forall_{x,y,z}.(x = y \wedge y = z) \rightarrow x = z$). Note that these expressions can be of any type, they do not have to be predicates and they certainly do not have to be valid.

hol_type An instance of this type denotes the type of a term. In the above example this would be a boolean, but many more types are allowed, such as function types, lists, sets and any user-defined type.

At this point, the distinction between the meta level and the object level becomes more clear: Instances of hol_type are types at an object level (types of terms), however, hol_type itself is a type at the meta level (a type defined in ML). For example, there is a function called 'type_of' that will take a term as input and will produce the type of that term as output. The signature of this function is term $\rightarrow$ hol_type and the result of the function on the expression above would be 'boolean', where 'boolean' is of type hol_type.

HOL makes use of many more types than just term and hol_type, but these are not directly relevant within the scope of this writing. More information can be found in the HOL4 description [27].

Being able to express terms in HOL, the next step is to prove them. There is a proof manager that can help in the process, or one can choose to do this interactively. Either way, the proof consists of telling the proof engine what to do. So called 'tactics' are the basic ingredient to do this. A tactic will tell the proof engine to try certain proof steps. This can range from the application of one simple rule, to a fully automated search of a proof. The choice of tactics is of crucial importance to the success and speed of the proof. There is a large number of built-in tactics in HOL and this base can be expanded by the user by defining new tactics.

When an expression has been proved to be valid, a 'theorem' will be produced by HOL. An instance of this type always denotes an expression (of type boolean) that has been proved to be a tautology[1]. Theorems can be stored for later use in other proofs.

From now on, we will call the expressions in the Meta language ML: ML statements. ML statements will be used to control the HOL engine or to gain more information about HOL terms. Anything to be used as 'valid' input to a

[1] There is one special exception to this. One might want to introduce theorems without proving them, for testing purposes. This is possible, but the theorems will be flagged. Any theorem that is proved using a flagged theorem will also be flagged. Using these flags, theorems that might not be sound can always be recognized.

HOL session has to be an ML statement, since ML is the interface between the user and HOL. In the following chapters, the distinction between ML statements and terms will become more clear when actually using them.

2.2.2 HOL Development

In the late 1970's, Robin Milner headed a team that developed the functional programming language ML. The main objective of ML was a system called LCF (Logic for Computable Functions), which was intended for interactive automated reasoning about higher order recursively defined functions. Although the original purpose of ML was clear, it incorporated a clear separation of meta language (ML) and logic (LCF), to be able to try other types of logic while using the same meta language. Milner published the ideas used in LCF on mechanizing formal proofs in 1979 [21].

The first version of the HOL system included an early version of ML and has been used for several years. The enhanced and rationalized version that was released in 1988 (HOL88) still used this earlier version of ML. In the 1990's, the development of HOL split up into a version developed by Konrad Slind, initially at the University of Calgary, which makes use of the last version of ML and a commercial version developed by ICL Secure Systems, which is based on the original version of ML. The last is now known as 'ProofPower' [3], while the first was released as HOL90. In 2004, an upgrade of HOL90 was released as HOL4, the version that will be the main proof engine in this project.

The primary application area of HOL was initially intended to be the specification and verification of hardware designs. However, the logic does not restrict applications to hardware. Since its start, the development of HOL has mainly focused on development of libraries. It is now widely used for commercial and academic purposes.

The development of HOL up to version 4 mainly focused on developing new libraries and extending existing ones. For HOL4, the main improvements were addition of the libraries BDDs, SAT and support for model-checking. The current development concentrates on linking the theorem prover to other programs. These include industrial CAD and CASE tools as well as a variety of automatic proof procedures. The long-term aim is to provide a flexible core proof engine that can be embedded in other CAD/CASE systems to provide a programmable platform for formal verification [1].

2.2.3 Meta Language

Although originally developed as a meta language for LCF, the language ML has now achieved status as a programming language in its own right. It is particularly known for its inference mechanism, that makes use of the Hindley-Milner type inference algorithm [25]. Using this algorithm, it is able to derive most types automatically, in contrast to languages such as Java and C++. This allows for easier and safe selection or distinction of theorems and formulae. Furthermore, it removes the burden of having to specify types when these can be deduced.

Even though several dialects of ML are known, a standard for the language was published by Milner [26]. This standard language is known as Standard

ML, or SML. The latest versions of LCF as well as HOL are based on SML. As default, HOL uses the Moscow [2] implementation of ML, which is a light-weight version, implementing the core SML language. If desired, other implementations of ML can be used along with HOL.

2.2.4 HOL Subset

The HOL engine itself does not have much built-in functionality. Its basis is rather small. It has been extended by a large number of libraries, that contain theories that support a variety of structures, from the logical 'and' to automated provers. In this section, only a little of this functionality will be explained. This is for two reasons. First of all because if new HOL functions are used in the thesis, they will be explained on the spot. Second, because the number of functions and syntax constructions that needs to be used from HOL is limited.

Types Types in HOL are rather complex. We will look into their definition and usages later. At this point it is useful to note that some types are predefined and can be used right away. These include `num` (an equivalent to the regular integer), `bool`, lists, sequences, etc. Others can be defined by the user through higher or lower abstraction level Meta Language functions.

Functions Functions in HOL (at the object level) are regular mathematical functions. At a low level of abstraction they are defined through theorems, but in this thesis we will always use a higher level method that does most of the underlying work for us. The method is called '`Define`' and will take a function name, parameters, types, function body and types in, among others, the following format:

```
Define 'functionName p1:T1 p2:T2 ... pn:Tn = E:Tn+1';
```

in which `p1` to `pn` are the parameters of the function and `T1` to `Tn` are their types. `E` is the expression that represents how the function should be evaluated (its body) and `Tn+1` is the result type of the function. We will look deeper into some of the details of `Define` later on.

Any function defined this way is a curried function. A call to it will therefore be written as:

```
functionName v1 v2 ... vi
```

In which `v1` to `vi` are the values on which the function is called. Each of these values should be of the right type (T1 to Ti respectively). The number `i` can be smaller than `n` from the definition. So for example, if we have the following definition:

```
Define 'sum x:num y:num z:num = (x + y + z) : num';
```

and we would execute the following expression:

```
sum 1 2
```

Then this would be perfectly valid. The result of the expression would be a function (num to num) that will add 3 (= 1 + 2) to its parameter.

Patterns Patterns are only supported in HOL in a limited number of places. Since these patterns are not directly useful to this thesis, we will not use any of them.

Proof The main goal of HOL is to find and prove expressions to be a tautology. An expression that has been proved to be a tautology using either axioms or previously proved statements is called a theorem. Except when testing, theorems cannot be inserted into the system without proving them. To prove one, at first we need an expression that results in a boolean value (a predicate). This can be entered into the HOL engine as a term. Then the proof can start. The topic of proof itself will be addressed later in this thesis. Once it has been proved, the term will become a theorem and can be stored for later use. All libraries of HOL are built in this way. They consist of many theorems, that all have to be proved to be able to use them. This typically happens during installation of HOL.

For obvious reasons, axioms in HOL do not have to be proved. The number of axioms used is small, in the most commonly used set-up just five. All theories are built on these by means of proof. Introducing axioms yourself is possible, but usually not required and it might break the logical basis of the system. Therefore, no axioms are introduced in this thesis, meaning that every theorem used can be proved from the four basic HOL axioms.

Chapter 3

A VDM to HOL translation

This chapter will focus on the different aspects involved when translating a VDM model into a HOL model. The topics covered are slightly broader than just a VDM++ AST to HOL AST translation, since also phases before and after the actual translation are included if these are of interest to the translation. Section 3.1 will look at the different steps and context of the translation. Section 3.2 will focus on what the source and target of the translation are, followed by the actual description of the translation in sections 3.3, 3.4 and 3.5. Section 3.5 will then complete the discussion of the translation by focusing on global problems that need to be solved during translation. In Appendix A, a complete description of the range of the translation can be found.

3.1 Translation strategy

The translation of a VDM++ model starts in general with a concrete VDM++ model and should usually end with a concrete HOL model. We can distinguish 5 steps:

1. *Parsing the concrete VDM syntax:* This will take a syntax-correct VDM document as input, producing a VDM abstract syntax tree (AST) as output.

2. *Type checking:* This will check for all statically decidable inconsistencies of the model (see Section 1.3) and makes sure all models are suitably typed to be handled by the HOL engine after having been translated.

3. *VDM AST to HOL AST translation:* The model from the previous step, will be transformed into a HOL-AST via a recursive descent strategy. This will use a compositional style: while decomposing the source AST, the target AST is composed. The similarity between the two trees make this possible.

4. *Post processing of the HOL AST:* Whenever the compositional style of the previous step is not sufficient, an additional post processing step is required. This step will involve several of those post processing steps. Examples are solving dependencies and substitution of synonym types.

5. *Generating the HOL concrete syntax from the HOL AST:* In a single descent of the HOL AST, the various HOL constructs in the tree can produce their concrete syntax, resulting in an 'executable' HOL model.

The first two steps are beyond the scope of this thesis, and will therefore only be discussed when their execution is required by one of the other steps. The other three steps constitute the contribution of this project to the translation. Steps three and four encompass the main translation issues. Step five is in most ways a simple code generation.

The discussion of the translation in this chapter will mainly look at the process as a whole. This means it will focus on a concrete VDM model to a concrete HOL model translation. When the separation of steps is of direct interest to the translation, a detailed description of this will be given. Furthermore, the concrete implementation of the two AST's are not directly relevant here and in most ways straightforward.

Looking at the overall translation process, there are three kinds of translations:

Types The translation of types will start with a VDM type and will result in a 'hol_type'. The brackets $\langle \ldots \rangle$ will be used to indicate a type translation. HOL_types are the main way of denoting types in HOL. More information can be found in Section 2.2.

Expressions The translation of expressions will start with a VDM expression and will result in a HOL 'term'. The brackets $\langle | \ldots | \rangle$ will be used to indicate an expression translation. HOL terms denote expressions in HOL. More information can be found in Section 2.2.

Definitions The translation of definitions will start with a VDM definition (of a function, type or class) and will result in one or more ML statements. These statements typically contain HOL types or terms, or references to declarations of these. The brackets $\langle \| \ldots \| \rangle$ will be used to indicate an expression translation. ML Statements are statements that can control the HOL engine. More information on these can be found in Section 2.2.3.

3.2 Translation restrictions

HOL has many restrictions on its models. These are mainly enforced by syntax regulations, but sometimes also restrictions that are only expressed at a semantic level (e.g. type unions in which the types have to be disjoint). When translating a VDM model to a HOL model, these restrictions cannot just be ignored. When dealing with a HOL restriction, it somehow needs to be avoided or solved during translation, or the restriction should be placed upon the input (VDM model) to the translation. Combining this with the fact that there is not enough time to translate the entire language of VDM, there are several restrictions on the VDM models entered into the translator. Below, the main restrictions are summarized and an explanation is given of each of these.

- *Functional* HOL is a functional language. Although there might be ways to translate an imperative-style VDM model and still be able to prove obligations about it, it is not within the scope of this thesis and is therefore left out. This means that all allowed VDM constructs are functional. No VDM statements will be translated. In the current implementation an error will be generated when a non-functional construct is being used. The translator will still try to translate the remaining (functional) part of the model.

- *Mutual recursive dependencies* Since HOL does not easily allow a forward declaration mechanism, mutual recursive dependencies can only be expressed in one definition. Mutual recursive dependencies over multiple definitions (like multiple types or multiple functions) are not allowed. In the current implementation an error will be generated during translation when there do exist mutual recursive dependencies in the input model. The translation will fail.

- *Disjoint type unions* HOL does not allow type union of non-disjoint types. Therefore also in the VDM model, only disjoint unions are allowed. Considering that this is hard to detect during translation, no error will be generated. HOL will eventually fail when using the model (although syntactically the model will be fine).

All three of these restrictions might be solvable in time. Due to the fact that solving them will simply take (much) more time than is available in this project, they have been left open.

3.3 Types

Types are one of the basic ingredients of most programming languages. VDM++ has a wide range and broad usage of types. The translation of them is therefore of crucial influence to the translation process. We can roughly distinguish two sets of types:

- *Basic types*, which contains all predefined types in VDM++ (integers, booleans, natural numbers, etc.) and most of the types that can be constructed (unions, lists, sets, etc.).

- *Advanced types*, which contains the quote type and the record type.

The two sets are disjoint and together they hold all VDM++ types available. Their distinction may seem somewhat arbitrary, but will become clear when discussing their translation in this chapter.

The first part of this section will focus on the translation of the basic types. The last part will focus on translation of the advanced types. In between, the management of types is discussed, which involves higher level topics that come into focus when trying to translate types (such as how to fit definitions into a model and how to deal with VDM++ type invariants). The information on type management is used in the explanation of the translation of the advanced types.

During the explanations in this section, in addition to the types, also some expressions may be translated if these are directly relevant.

3.3.1 Translating basic VDM++ types

We will look at two sets of basic types: the built-in types and the constructed types. The first group holds all types that have been predefined in the VDM++ language and require no further construction from the user to be able to use them (e.g. integers and booleans). The second group involves types that can be constructed using type constructors and are usually slightly more complex than the first group (e.g. sets and unions). The following two subsections will focus on the translations of types in each of these two groups respectively.

3.3.1.1 Basic built-in types

All of the basic built-in types included in the VDM++ subset have synonyms in HOL. Translation is therefore straight forward. The different types and their translations are listed in Table 3.1. Added to these are the translations of values within each of the types. The variable x in the last column is supposed to be replaced by an arbitrary VDM value of the specific type.

type	$\langle type \rangle$	Translation of VDM value x
token	ind	x
boolean	bool	$\begin{cases} T & x = true \\ F & otherwise \end{cases}$
char	char	$''\#x''$
nat	num	x
real	real	x

Table 3.1: Translations of basic built-in types. The first column contains the type being translated, the second its translation and the last column contains the value translation.

3.3.1.2 Basic constructed types

In addition to the fixed basic types, a vast number of types can be constructed using type constructors. Just as in the case of the basic types, these have a simple translation to HOL, usually no more than reordering some identifiers, or using different names. This section discusses the translation of each of the basic constructive types and their values individually. If useful, additional operators are discussed, but many of them are left out. A complete discussion of operators can be found in Appendix B.

Product Translation of the product type is as follows:

$$\langle T_1 * T_2 * \ldots * T_n \rangle = \langle T_1 \rangle * \langle T_2 \rangle * \ldots * \langle T_n \rangle$$

Translation of a product value is similar:

$$\langle |\text{mk\_}(T_1, T_2, \ldots, T_n)| \rangle = (\langle |T_1| \rangle, \langle |T_2| \rangle, \ldots, \langle |T_n| \rangle)$$

List & Set Translation of the ordered and unordered collections (list and set) are very similar. The translation of the types look like:

$$\langle \text{seq of } T \rangle = \langle T \rangle \; list$$

$$\langle \text{set of } T \rangle = \langle T \rangle \; set$$

The translation of their respective values look like:

$$\langle | [T_1, T_2, \ldots, T_n] | \rangle = [\langle | T_1 | \rangle, \langle | T_2 | \rangle, \ldots, \langle | T_n | \rangle]$$

$$\langle | \{T_1, T_2, \ldots, T_n\} | \rangle = \{\langle | T_1 | \rangle, \langle | T_2 | \rangle, \ldots, \langle | T_n | \rangle\}$$

In which the order of T_1 to T_n on the right-hand side is not relevant in the second translation, but is relevant in the first (because of the difference between the unordered and ordered collection).

Map Translation of the map type looks like:

$$\langle \text{map } T_d \text{ to } T_r \rangle = (\langle T_d \rangle | - > \langle T_r \rangle)$$

Each map is considered to be a finite function in VDM++, but is considered to be a regular value (not a function) in HOL. This incompatibility provides some problems when translating the application of maps. This topic will be discussed in Section 3.4.5.

Map instantiations can not be translated in a straightforward way. In VDM++ these can be written down in a similar way as the sets and lists:

```
{1 |-> 2, 2 |-> 3}
```

Which produces an integer to integer mapping which will map `1` to `2` and `2` to `3`. In HOL this enumerating method does not exist and a slightly more extensive approach has to be used. The idea is to start with the empty mapping (a constant in HOL) and update this with any additional mapping desired. The above example would then become:

```
FEMPTY |+ (1, 2) |+ (2, 3)
```

The `|+` is the map update operator and `FEMPTY` is the empty mapping. Note that each of the updates use a product value (`(1,2)` and `(2,3)`) to indicate the new mapping. In short, the translation can be described by:

$$\begin{aligned} &\langle | \{T_{1a} \text{ |-> } T_{1b}, \ldots, T_{na} \text{ |-> } T_{nb}\} | \rangle = \\ &\text{FEMPTY } |+ (\langle | T_{1a} | \rangle, \langle | T_{1b} | \rangle) \; |+ \; \ldots \; |+ (\langle | T_{na} | \rangle, \langle | T_{nb} | \rangle) \end{aligned}$$

The map translation assumes that only correct mappings are being translated. So for example the type incorrect mapping

```
{2 |-> 2, 2 |-> 3}
```

in which 2 is mapped to two different values, would yield a translation which is not semantically equivalent, namely one with only one mapping for two, as the first will be overwritten. A proof obligation should be generated to prevent this kind of incorrect translation. This proof obligation is not allowed to use the map enumeration itself, as its translation is incorrect. In most cases this will indeed hold, but it in the extraordinary case that the proof obligation does use the expression (which is when one of the source or target expressions directly or indirectly refers to the function in which the cases expression is used) and furthermore the proof obligation is not valid, the translation might be incorrect. No error will be generated during translation if this occurs, but the scenario is extremely rare.

Function Function types can be translated easily, as they are equivalent in VDM++ and HOL:

$$\langle T_1 * T_2 * \ldots * T_n\text{->}T_{n+1}\rangle = \langle T_1\rangle * \langle T_2\rangle * \ldots * \langle T_n\rangle\text{->}\langle T_{n+1}\rangle$$

3.3.2 Managing types

Having derived the translation for most of the types used in a VDM++ model, the next step is to actually use these translations. Consider a small piece of a VDM++ model:

```
types:
  bitString = seq of bool
functions:
  flip : bitString -> bitString
  flip (s) ==
    if s = [] then []
    else [not(hd(s))] ^ flip(tl(s))
```

This will define the type `bitString` and a function `flip` on it, that will flip each of the bits in the given string. The `^` operator is a sequence concatenation and the function `not` is equivalent to the logical negation. When translating this model to HOL, we can easily find a translation for our sequence of booleans type. Using the previously defined translations, this would become "`bool list`". However, the translation of the usage of the type `bitString`, as well as its definition raises the questions of how to *define* types in HOL and how to use these.

3.3.2.1 Type definitions in HOL

Before looking at the translation itself, let us take a look at the ways types can be defined in HOL, in order to understand the ideas behind the translation. There are three main strategies in HOL when defining types [27]:

Advanced type definition Mentioning the advanced strategy first may seem somewhat out of the ordinary, but it is in fact the easiest method to use. HOL has a built-in function '`hol_datatype`' that can define rather advanced 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 will produce the type definition as a result, as well as several proved theorems that ease the use of it (for example an induction theorem). `hol_datatype` can mainly handle records, tree structures and enumerations. All basic types that might result from the translations defined in 3.3.1 can be used in a definition.

At first sight this method seems to be more than sufficient on its own, to use for any kind of type definition. Unfortunately, the function is only meant to deal with 'data' types. This means that it is not able to handle simple type definitions. Suppose I would like to define a different name for integers (a simple synonym definition), then the syntax of this function simply would not allow me to. Furthermore, adding invariants to types (like the ones explained in 2.1.5.1) is not an option.

Type definition using an existing type The main means of defining types in HOL is by using an existing type. Instead of giving an explicit description of what values belong to a certain type, it is also possible to take an existing type and select a subset of values out of it, thus indirectly defining what the values of the new type should be. Suppose I want to define a new type smallInt, which contains all integers smaller than 10, then I could choose the integer as existing type to use and specify that I would like to use all integers smaller than 10.

What makes this method slightly more complex is that it is not directly possible to use the new values, since they do not have any names. In the example of smallInt, if one wants to use the number 1 in smallInt and would write 1 trying to indicate it, He would not be referring to the right number. The number 1 actually refers to its instance of type integer, just like it always has, not the one from smallInt. Therefore, two additional functions are required: one that will translate values from the existing type to the new type and one that will do exactly the opposite. These functions are also known as the Abstraction and Representation functions. HOL will define these automatically.

At first thought, not all types may seem definable by using existing types. But the crux is in what types are available. Apart from the regular types like boolean and integers, there is one rather special kind of type, called `ind` (short for induction). Any value belongs to this type. Or in other words, the values belonging to any arbitrary type are a subset of `ind`. The result is that any type can be described by restricting the `ind` type.

A disadvantage of using this method, is that it requires a proof that the type that is being defined is not empty (contains at least one value). For humans this is easy for most definitions. In the case of smallInt, this would be "Try to find an integer that is smaller than 10". Not very complex. But trying to find this value automatically can be extremely hard and can be reduced to an extended satisfiability problem (of the invariant), which in general cannot be solved efficiently.

No type definition When using no type definition at all, clearly, the types will not be defined, but it certainly is a strategy of tackling the definition problem. Many problems that arise when really defining types could be avoided this way. HOL provides a means called type variables that can support this. Type

variables are exactly what their name indicates: variables that can contain any type assigned to them. So when I define a type variable `v` to contain the type `integer`, I can replace any reference to the type `integer` by the variable `v`. A VDM++ type definition could be translated to an assignment to a carefully chosen type variable. All we have to do next is substitute each of the usages of the types, by a reference to the right variable.

Unfortunately, the same problem as with advanced type definition occurs here: no invariants can be stated on the types. Furthermore, the method is rather user-unfriendly, as all types are declared as variables. This does not only make the resulting model hard to read, it also holds the danger that a variable is changed[1].

3.3.2.2 Translating type definitions

Since each of the type definition methods in HOL has its advantages and disadvantages, it is hard to choose the best option. A first attempt was made to define all types by means of existing types. This allowed for easy incorporation of invariants and easy integration into the model (no names have to be changed and no additional definitions were required). Unfortunately, the proof of each of the types being non-empty proved to be rather problematic. Apart from the fact that it is hard to prove automatically, it sometimes is not even true. Types in VDM are allowed to be empty.

Mainly due to the difficulty of proving non-emptiness without user interaction, this method was discarded. The current translation uses a combination of the other two methods. Before looking at it, let us take a look at what a VDM++ type definition looks like exactly:

```
typeName = typeDescription
inv v == typeInvariant;
```

The type description is in the format of any of the descriptions already discussed and translated in Section 3.3.1 (e.g. `integer`, `boolean | integer`, or `integer * integer`). The type invariant is a predicate on `v` [2]. We will ignore the invariant for now, as this will be discussed in the next section. The remainder of the definition thus consists of a name and a description.

We can split up all VDM++ type definitions in the following two groups:

1. Record types & Quote types: the advanced types
2. Other (tuples, unions, enumerations, synonyms, etcetera): the basic types

The first group is special in the sense that it can be handled by the advanced type definition method discussed in the previous section. As this method introduces some very useful theorems along with the definition (such as theorems to use for induction), it is used for all type definitions belonging to group 1. Usage

[1] In addition to these disadvantages, there are also some slightly more technical issues to overcome when using this method. These are all related to the separation of levels in HOL. The Meta-level, at which the variables exist and the Object-level at which the model and all proofs exist. A sufficient cooperation between items on these two levels is hard to achieve, especially when tactics come into focus.

[2] `v` may be a pattern. The translation of those is discussed in section 3.4.6.

of this method is relatively straightforward. The syntax of VDM++ has to be rewritten to the syntax specific for the `hol_datatype` function of HOL, but that is no more than replacing some characters. For example the VDM++ record definition of a two-dimensional integer coordinate:

```
coordinate :: x : int
              y : int;
```

would be translated to the following expression:

```
hol_datatype ('
  coordinate =
    <|
       x : num ;
       y : num
    |>
');
```

The <| and |> signs indicate a record and the word `num` is the name of the HOL type, which is equivalent to the integer in VDM++. Definitions of records and quotes are discussed in more detail in Section 3.3.3 on advanced types.

The second group cannot be translated using the advanced data type definition. Furthermore, the definition by using an existing type proved to be too difficult to use automatically, which leaves us with the third option. The biggest disadvantage of this option is using all the variables. Therefore, a similar idea is implemented that is enforced during translation. Each of the type definitions in a VDM model consists of a name and a type description (ignoring the invariant for now). The meaning of this is that the type name is considered to be equivalent to the type description and thus, each occurrence of the type name can be replaced by its description. Which is exactly what happens during translation. A list of all declared types is maintained, containing the names and the corresponding type descriptions. Each occurrence of a type name will be replaced by its type description. So instead of translating the following model:

```
types:
  coordinate = int * int;
    -- (A coordinate definition using a product type)
functions:
  sum : coordinate * coordinate -> coordinate
  sum (c1, c2) ==
    ....
```

theoretically, the following model is translated:

```
functions:
  sum : (int * int) * (int * int) -> (int * int)
  sum (c1, c2) ==
    ....
```

which has any occurrence of `coordinate` replaced by its description `(int * int)`. In practice this replacement happens directly after the translation, but the result would be similar to translating the above model.

3.3.2.3 Type invariants strategy

In addition to a plain type definition as translated in the previous section, VDM++ holds the possibility to enrich the new type with an invariant. This invariant will restrict the number of values that belong to the type and will therefore significantly change the meaning of it. Take for example the following piece of a VDM++ model:

```
types:
  specialNat = nat
  inv x == x <> 2
functions:
  getSmallAndSpecialNats : () -> set of specialNat
  getSmallAndSpecialNats () ==
    {i | i : specialNat & i < 5}
```

Since nat represents the natural numbers, execution of the `getSmallAndSpecialInts` would result in the set {0, 1, 3, 4}. Element 2 is left out, since it does not satisfy the invariant and is therefore not a member or the specialNat type. If we would have left out this invariant, the result would have been {0, 1, 2, 3, 4}. So invariants can change the semantics of the model.

As we have seen, out of the three discussed type definitions in HOL, only one was able to handle invariants directly. Unfortunately, this method was not sufficient for any of our type definitions. We will therefore have to accept that the types cannot be defined exactly as they were in the VDM++ model and use a different approach. A first thought might be to simply state the invariant in HOL, simply saying that all values of a type satisfy the invariant. However, how to say this is of course the question. Introducing it as a theorem is not an option, as it cannot be proved (any value that does not satisfy the invariant would be a counterexample of the theorem). And introducing it as an axiom (which does not have to be proved) is not an option either, as this would immediately introduce a contradiction, since there would then be one axiom stating that a certain value belongs to a type and another, for the invariant, stating that it does not.

The option left is just to leave them out. During definition, this is of course easy, but in some way, the remainder of the model has to be altered to reintroduce the invariants. After all, we do not want to change the semantics of the model. Leaving invariants out is the option used during translation. The key is to take a look at all of the locations where a type that has an invariant can occur and make sure that at these locations, the invariant is reintroduced. As these locations are mainly parts of expressions, it is not a topic of this section. The translations of expressions will have to be explained first. The reintroduction of invariants is therefore discussed in Section 3.5.2.

3.3.3 Advanced types

As explained in the previous section, there is a function in HOL that eases the process of defining 'data' types. It requires the name and a description of the type as input and will define the type along with several supporting theorems as a result. This section will explain definition and usage of the types that can

be defined using this function.

In contrast to the types explained earlier, using this 'data' type function makes sure that the type is actually defined as a type in HOL. As a result, we are also allowed to use the types just like they were used in the VDM++ model. No need to substitute anything. As long as the names of the types are kept equivalent (which they are in the current implementation), a usage of a type name in the VDM++ model can be copied into the HOL model without changing anything:

$$\langle \text{advancedTypeIdentifier} \rangle = \text{advancedTypeIdentifier}$$

3.3.3.1 Quote type definition

As seen before, quote types are types that only contain one value. This value is the type name written between < and >. So if we had defined a quote type named `Red`, then the only value of `Red` is `<Red>`. These types are different from other VDM++ types, in that they do not have to be defined prior to using them. For example, we could write the following VDM++ model:

```
types:
  Green = <LightGreen> | <DarkGreen>
  Red   = <LightRed> | <DarkRed>
  Color = Green | Red | <White>
```

This shows three type definitions. The first two (`Green` and `Red`) are in most programming languages considered to be enumerations. However, in VDM++ these two statements imply four more definitions, namely, the definitions of the types `LightGreen`, `DarkGreen`, `LightRed` and `DarkRed`. Each of them containing one single value. The third definition in the code (the one of `Color`) is a regular union of the two previously defined types and an additional value `<White>`, which implied the definition of `White`.

This possibility in VDM++ to use values of quote types without defining them does not exist in HOL. In HOL, all values have to be declared before using them. Therefore, all definitions which are implicit in the VDM++ model will have to be made explicit.

Quote values in VDM++ can be used in types and in expression (which basically encompasses most of the subset we are discussing). Each usage implies a type definition. Type definitions cannot be stated in type definitions or in expressions. Type definitions will therefore have to be included just before the definition or expression in which the value is used.

So if we look at the example above, we would first have to define the two quote values for the shades of green:

```
HOL_datatype ‘LightGreen = LightGreenQuoteLiteral‘;
HOL_datatype ‘DarkGreen  = DarkGreenQuoteLiteral‘ ;
```

These two lines will define the types `LightGreen` and `DarkGreen`, each containing one single value: `LightGreenQuoteLiteral` and `DarkGreenQuoteLiteral` respectively. The names of the values are not equivalent to the ones in the VDM++ model (`<LightGreen>` and `<DarkGreen>`), since the characters < and

> can not be used in identifiers. So a slightly different, but unique, name is chosen using the suffix "`QuoteLiteral`" and additional numbers if required. Then we get to the point where we would be translating the definition of the type `Green`. However, `Green` is simply a union type (not a quote type) and as we have seen in Section 3.3.2.2, these definitions are not being translated into a HOL definition. Their definition will be left out of the HOL model and a substitution based translation is used if the type names are being referred to.

Translation of the definition of `Red` is similar to that of `Green`:

```
HOL_datatype `LightRed = LightRedQuoteLiteral`;
HOL_datatype `DarkRed  = DarkRedQuoteLiteral` ;
```

And finally the definition of `Color` involves the implicit definition of `White`:

```
HOL_datatype `White = WhiteQuoteLiteral`;
```

Since the types `Green`, `Red` and `Color` have not been defined, any usage of these types require some substitutions. Suppose we would be using the type `Color` somewhere, then as a first step, this would be replaced by:

```
Green + Red + WhiteQuoteLiteral
```

The plus sign is the HOL equivalent for the type union used in VDM++. The value `WhiteQuoteLiteral` is the single value of the type `White` that we defined above. The types `Green` and `Red` are not defined either and therefore require an additional substitution resulting in:

```
(LightGreenQuoteLiteral + DarkGreenQuoteLiteral) +
(LightRedQuoteLiteral + DarkRedQuoteLiteral) +
WhiteQuoteLiteral
```

Which is the final result of the type translation.

3.3.3.2 Record type definition

There are two main difficulties to overcome in the translation of record types:

1. In contrast to HOL, VDM record instances have a tag indicating their type.
2. VDM assumes an ordering in the record fields, while HOL does not.

The additional tag in records in VDM has one problematic side effect, namely that equality is also based on the type. In HOL, two records are equal if they have the same fields and the values for each of the fields are pairwise equal. In VDM they must also be of the same type in addition to this. To solve this during translation, an extra field is added to the HOL record type, containing the tag (the records' type name) as a string. Equality then automatically includes the check of tag equality.

The ordering of the fields in VDM is more complex. During definition of a record, this ordering is not required, as we translate from an ordered definition in VDM to a non-ordered definition in HOL. However, the missing ordering will give us problems when constructing records or when selecting fields out of a

record based upon their number in the ordering[3]. To solve this, we will introduce additional functions into the HOL model when we know the ordering, namely during definition. These additional functions include a record constructor for each record type and a field selector for each field in a record type.

In the following example, the same coordinate definition is used as in 3.3.2.2:

```
Coordinate :: x : nat
              y : nat;
```

This will translate to:

```
HOL_datatype ‘Coordinate = <| x:num;
                              y:num |>‘;
Define ‘make_Coordinate (x_recConstrParam:num) (y_recConstrParam:num) =
  <| recordTag := "Coordinate" ;
     x := x_recConstrParam ;
     y := y_recConstrParam |> ‘;
Define ‘Coodinate_field_1 (record:Coordinate) = record.x‘;
Define ‘Coodinate_field_2 (record:Coordinate) = record.y‘;
```

The first statement is the definition of the type, the second the constructor definition and the third and fourth are the field selectors. Note that the parameters used in the constructor have complex names, to make sure they do not equal any of the record field names. The reason that the constructor is called make_... instead of mk_... (which would be following the VDM standard) is that mk_... is already defined in HOL when introducing records and can therefore no longer be used. HOL uses this as a constructor, where the order of fields is of no importance and therefore useless in this situation.

The definition of this translation, for any record type name rtn and number of fields n is:

$\langle \text{rtn} :: f_0 : T_0\ f_1 : T_1 \ldots f_n : T_n \rangle := \text{constructorDef};\ \text{fieldSel}_1; \ldots \text{fieldSel}_n$

in which: (recConstrParam is abbreviated to rcp):

constructorDef :=

$$\text{Define ‘make\_rtn } (f_0\text{\_rcp} : T_0) \ldots (f_n\text{\_rcp} : T_n) = \\ < |\text{recordTag} := \text{”rtn”};\ f_0 := f_0\text{\_rcp}; \ldots; f_n := f_n\text{\_rcp}| >$$

and for any i from 0 to n:

$$\text{fieldSel}_i := \text{Define ‘rtn\_field\_i (record : rtn)} = \text{record}.f_i\text{‘};$$

3.4 Translating expressions

Using the types of the previous section, this section will focus on the translation of expressions. Each of the following subsections is devoted to one kind of expression.

[3]This will occur in patterns. An explanation of these can be found in 3.4.7

3.4.1 Identifiers

The most common type of expressions in VDM models is the identifier. Identifiers are regular variables or constant references. We can distinguish two kinds of identifiers: user-defined identifiers and 'built-in' ones. The first are the regular variables and constants that are defined during execution of the model. The second are the pre-defined constants, that are available during execution of any model (e.g. the numbers, characters or Boolean values). We will call these built-in constants 'literals' instead of identifiers from now on. Note that the keywords in VDM are distinguished from identifiers by the parser. VDM keywords no longer exist as identifiers in the VDM AST.

Custom VDM identifiers are translated to HOL, by simply copying their name:

$$\langle|\text{Identifier}|\rangle = \text{Identifier} \tag{3.1}$$

This translation is based on the assumption that all identifiers will be bound in the same way as they were in the original model. We will show throughout the rest of this chapter that this will indeed be satisfied.

A note worth making at this point, is that this type of translation requires that the same identifier names are allowed and that they may be used in the same way. The first indeed holds, because the set of allowed identifier names is larger in HOL than it is in VDM, however the second will not be satisfied when dealing with quote literals. That is why these have to be translated differently from Equation 3.1. Their translation has been explained in Section 3.3.3.

The translation of literals has been explained for each of the native types in 3.3.1.1.

3.4.2 Conditional

A basic and straightforward expression used in VDM is the conditional, or if-then-else. Since HOL also has a conditional as part of its language the translation is as follows:

$$\langle|\text{if } P \text{ then } E_1 \text{ else } E_2|\rangle = \text{if } \langle|P|\rangle \text{ then } \langle|E_1|\rangle \text{ else } \langle|E_2|\rangle$$

3.4.3 Apply expression

Apply expressions in VDM are basic function applications. The same holds for HOL. A difference is in the functions being curried or not. HOL functions always are, while VDM functions might not. However, this does not provide any difficulties, as the use of non-curried functions will automatically be 'simulated' by a curried function. Or in other words: The application of a non-curried function f on a list of parameters would be equal to the application that one would use when f *would* be curried. So for any expressions $E_{...}$ and number of parameters n we get:

$$\langle|(E_f\ E_{P_1}, \ldots, E_{P_n})|\rangle = \langle|E_f|\rangle\ (\langle|E_{P_1}|\rangle) \ldots (\langle|E_{P_n}|\rangle)$$

The first expression, E_f, has to result in a function, in order for the expression to be type consistent. Also the types of the other expressions have to be consistent with the argument types of this function. As mentioned earlier, a type checker should verify this in advance of the translation.

3.4.4 Operators

There is a large number of operators in VDM. Each of these is no more than an expression resulting in a function. The translation of each of the operators is listed in Appendix B. There is a distinction between unary and binary operators. There are no ternary operators in the VDM language (although constructs like the basic conditional might be considered as ones, but these are called differently). In translation, the operators are substituted for the translations as listed in the appendix.

3.4.5 Map application

The most used operation on maps is application. By providing a map and a value in its domain, the application will result in whatever value is associated with the given domain value in the map. Maps in VDM++ are considered to be regular finite functions. Functions from their domain to their range. Map applications are therefore no different from function applications. The following expression would locally declare a mapping and apply it to the value 2:

```
let
  map = {1 |-> 2, 2 |-> 3}
in
  map(2)
```

Since 2 is in the domain of the mapping, the expression will result in the associated value 3. If 2 would not have been in the domain, the expression would have been undefined.

In HOL maps are not directly considered to be finite functions. They still are finite and in some ways can be handled similar to functions, but their application cannot be written as a function application. Instead of the previous VDM++ expression, we would write in HOL:

```
let
  map = FEMPTY |+ (1, 2) |+ (2, 3)
in
  FAPPLY map 2
```

This involves two major changes: The first change is that the declaration of the map is written using the empty map (as discussed above). The second change is in the application. We cannot directly apply `map` to `2`, since `map` is not a function. We use a dedicated function instead. This function (`FAPPLY`) will select the sought value from the mapping and return it as result.

This translation from the function-like map application in VDM++ to an indirect map application in HOL seems sufficient at first thought. There are however two issues left unsolved: What if a variable that is either a function *or* a map (a union type) is applied to a value. In VDM++ this would be perfectly valid. At run-time the right application (function or map) would be selected. In HOL this would mean that at run-time, depending on the real type, either a function application has to be performed, or a function has to be called to do the map application. This is simply not possible in HOL. The second

issue is, that even if the first issue would be ignored, the knowledge of which of the applications in a VDM++ is a map application and which are function applications is missing. Type information is not available currently, so we will have to do without.

3.4.6 Patterns

Although patterns are not expressions themselves, they can occur in many locations in VDM constructs. In HOL very few patterns are allowed and we will use none. A translation would therefore consist of replacing all patterns with semantically equivalent HOL expressions. Since this process is rather complex to do in one step, we will use a different approach that will be used several times again in other parts of the translation. The VDM expression will not directly be translated to HOL expressions, but only rewritten to VDM expressions. These VDM expressions can then be translated in a next step. If we rewrite it to VDM expressions for which we already have a translation defined, we can even reuse code and save a lot of work.

VDM has several usages of patterns, depending on the location in which the pattern is used. Examples are patterns in assignments, patterns as parameters to functions, patterns used in comparisons, etcetera. Each usage requires its custom translation. Most of them only being slightly different from the other. To prevent having to define a different translation for each usage, all pattern translations are being rewritten to the two possible pattern usages mentioned below. These two usages are VDM like, but not always according to the VDM++ language description. But since these expressions can still be translated to HOL, it does not have to be.

- *Assignments to patterns* For example $\text{mk\_}(x, y, z) = \text{mk\_}(1, 2, 3)$ should result in x being 1, y being 2 and z being 3. The right side is always an expression, while the left side is a pattern. A type checker has to verify in advance that the sides are compatible, such that inconsistent situations like $\text{mk\_}(x, y) = \text{mk\_}(1, 2, 3)$ are prevented.
- *Equality of expression and pattern* These are predicates containing a pattern as left-hand side of an equality and an expression as the right-hand side. For example: $\text{mk\_}(2, 1, 3) = \text{mk\_}(1, 2, 3)$, would yield false. In fact, the pattern given earlier, $\text{mk\_}(x, y, z) = \text{mk\_}(1, 2, 3)$ is also a valid equality pattern, except that the meaning is slightly different: When variable occur in this type of pattern, they must always be bound. Free variables are not allowed, as it would be impossible to calculate the equality without knowing their values.

In regular VDM, there is only one kind of pattern evaluation. This will result in the same way as the first usage above, with an extra option to be undefined if the second usage would result false. Since being undefined is not an option in HOL, the split of these will simulate this.

The rewriting of these two ways of using patterns is very similar. We will give the rewrite rules of the first and describe how to derive the rewrites of the second using the first. To distinguish the pattern translation from the regular translation of expressions, we will use new bracket symbols, $[\![\ldots]\!]$, indicating

this specific rewrite function (different from the three translation functions we have seen so far).

There is a limited number of possible patterns. From these, the subset given below is supported by the translator. The rewrite for each type of pattern is given for the 'assignment rewrite':

- *Identifier pattern* This type of pattern in an assignment represents a regular expression to variable assignment. The type of assignment one would see in most programming languages. This also includes HOL. Rewriting is therefore easy:

$$[|ID = E|] = ID = E$$

- *Don't care pattern* As the name of the pattern proposes, the value of the right-hand side is of no relevance to the model. We can therefore just ignore it. To do this and still make sure it is a valid rewrite, we can use several ways. For simplicity, it is translated to a useless assignment here. In practice, the entire statement will be left out.
  Assuming $ignoreID$ is a unique variable in the model, the translation can be:

$$[|- = E|] = \text{ignoreID} = 1$$

- *Record pattern* Record patterns are constructive patterns, in that they can contain other patterns themselves. The rewrite uses an inside-out approach: All patterns inside the record pattern must be matched with the expression outside the pattern. To solve this, all these patterns are taken out of the record and stated individually. For a record tag R and its n fields, we get:

$$\begin{aligned}[|\text{mk\_R}(p_1, \ldots, p_n) = E|] \quad = \quad & [|p_1 = \text{R\_field\_n}(E)|] \text{ and } \ldots \\ & \text{and } [|p_n = \text{R\_field\_n}(E)|]\end{aligned} \tag{3.2}$$

  The R_field_i functions in 3.2 refer back to the functions defined during the type definition of the record. They simply select field i from the record. More information on these can be found in Section 3.3.3.2.

- *Tuple pattern* The tuple pattern is very similar to the record pattern and rewritten in exactly the same way:

$$[|\text{mk\_}(p_1, \ldots, p_n) = E|] = {}_1 = E.\#1|] \text{ and } \ldots \text{ and } [|p_n = E.\#n|]$$

  In which $\#i$ is a HOL function that will select the i'th field from the tuple.

Using these individual rewrites, we can translate any combination of these. An example of such a combination is the following assignment pattern:

```
mk_(-, x, mk_R(y, -, z)) = tuple
```

In this example, the variable tuple is a 3-tuple, which has a record of type R as its third field (these facts should have been verified by the type checker). The first step of rewriting will be the rewriting of a tuple pattern, with the following result:

```
[|             - = tuple.#1 |] and
[|             x = tuple.#2 |] and
[| mk_R(y, -, z) = tuple.#3 |]
```

The second step is a rewriting of the three individual patterns (don't care, identifier and record). The result will look like:

```
ignoreID1 = 1                       and
        x = tuple.#2                and
[|      y = R_field_1(tuple.#3) |] and
[|      - = R_field_2(tuple.#3) |] and
[|      z = R_field_3(tuple.#3) |]
```

And finally, rewriting the last three patterns, this becomes:

```
ignoreID1 = 1                       and
        x = tuple.#2                and
        y = R_field_1(tuple.#3)     and
ignoreID2 = 1                       and
        z = R_field_3(tuple.#3)
```

Using the implemented code, the ignoreID's will be left out.

The equality of expression and pattern is rewritten almost equivalent to the assignment to a pattern. The difference is mainly in the used *and* in the tuple and record pattern. This should be a logical and ($/\backslash$) when translating it as a predicate. Apart from that, the useless assignment in the don't care pattern either has to become true (T), or it has to be left out. The function that is obtained in this way will be denoted as $[\!|| \ldots ||\!]$

Needless to say, in the implementation of this type of pattern translation mainly uses the same code as the assignment pattern does.

Pattern types that are not supported are the union pattern and the concatenation pattern.

3.4.7 Let expressions

Just as VDM, HOL has a let expression defined in its libraries. Translation of the let expression itself is therefore easy. The patterns in it however make it more complex. The definition of the translation is:

$\langle\!|\text{let } p_1 = E_1, \ldots, p_n = E_n \text{ in } E|\!\rangle = \text{let } [\!|p_1 = E_1|\!] \text{ and} \ldots \text{ and } [\!|p_n = E_n|\!] \text{ in } E$

An example will show this definition in practice:

```
let
  mk_(x, -) = mk_(1, 2)
in
  mk_(x, x)
```

Will be translated to:

```
let
  x = 1
in
  (x, x)
```

The let expression is a basic building block for most other operators that have to deal with patterns. As mentioned before, all of these should be rewritten to either a pattern in an assignment, or a pattern in an equality. The first is nearly always a let expression, since this is the only obvious way to use an assignment in a functional subset. The pattern that results from the translation of a let expression can therefore be seen many times in the translation of a regular VDM model. Even if the model itself has no explicit let expressions at all.

3.4.8 Cases expression

The cases expression is like the conditional, but has a crucial difference: Patterns are allowed in the matching values of a conditional. Thankfully, we can use our regular pattern translation to solve this. In fact, this is exactly the pattern-expression equality we have discussed in 3.4.6. The translation of a cases expression with n cases (excluding the 'others') consists of:

If $n = 1$

$$\langle|\text{cases } E: \ p_1 -> E_1, \text{others} -> E_o \text{end}|\rangle = \\ \text{if } [\|p_1 = E\|] \text{ then } E_1 \text{ else } E_o$$

Otherwise

$$\langle|\text{cases } E: \ p_1 -> E_1, \ldots, p_n -> E_n, \text{others} -> E_{n+1} \text{end}|\rangle = \\ \text{if } [\|p_1 = E\|] \text{ then } E_1 \\ \text{else } \langle|\text{cases } E: \ p_2 -> E_2, \ldots, p_n -> E_n, \text{others} -> E_{n+1} \text{end}|\rangle$$

In which $\langle\|\ldots\|\rangle$ is the pattern equality translation as defined in 3.4.6.

The above translation assumes the 'others' clause to be mandatory, since it is an expression that is not allowed to be undefined. But in VDM, the others clause is allowed to be left out. If this is done, an extra proof obligation will be generated, that states that all other cases together must be total, solving the undefinedness problem at a different location. Any such let expression that has no otherwise clause can be rewritten to one with an otherwise clause, by changing the last clause to be the otherwise. However this does assume that the proof obligation is valid. A proof should eventually take care of this, but in order for the proof to be valid, it cannot use this cases expression itself (since it has to be proved to be total first). In most situations the proof obligation will not use the cases expression. In the extraordinary case that it does use the expression (which is when one of the conditions directly or indirectly refers to the function in which the cases expression is used) and furthermore the cases are not total, the translation might not be valid. No error will be generated during translation if this occurs, but the scenario is extremely rare.

3.4.9 Quantifiers

A VDM++ quantifier typically looks like:

```
forall bindList & predicate
```

or

```
exists bindList & predicate
```

in which the bindList can contain either a type bind:

```
pattern : type
```

or a set bind:

```
pattern in set someSet
```

The first binding says that the pattern must be of type `type`. This means that all identifiers in the pattern are bound to a type. For example:

```
mk_(x, -, y) : int * int * bool
```

would bind x to int and y to bool. The set binding binds the pattern to all values in a set in a similar way as with types. The difference is that a set is finite in VDM++, while a type does not have to be.

In HOL, the quantifier format that we will be using is much more basic. There are no pattern 'features' included like we have in VDM++. In fact, not even a dedicated binding mechanism will be used. The quantifier expression looks like:

```
! variableList . predicate
```

for the universal quantification over the variables in the variableList, or

```
? variableList . predicate
```

for the existential quantification.

As the source and the target of the translation are clear, the quantifier translation itself is the next topic. In this translation, basically two difficulties meet each other. The first is the translation of patterns and the second is the translation of a bind itself. Unfortunately these two difficulties cannot be dealt with separately. In order to make the translation, we will have to look at the combination of them. Before we can do this, an additional function will have to be introduced, that will be used in the translation:

Get identifiers from a pattern The key elements of a pattern are its identifiers (or also called identifier patterns). These are the local variables that will be filled in a let expression, or the variables that will eventually be compared when looking at a pattern-expression comparison as explained in Section 3.4.6. For reasons that will become clear later, we need to be able to extract these identifiers from a pattern. This is a rather straightforward procedure. It might be done by simply traversing the pattern from left to right and collecting each of the identifiers met, or it might be done by a recursive descend into the pattern, of which the identifier parts (always leafs) are collected. Either way, we will call this function `getIdentifiers`, which will give us a set of identifiers when a pattern was given as input. Some examples of its functionality are:

```
getIdentifiers(-) = {}
getIdentifiers(a) = {a}
getIdentifiers(mk_(a, -, 1)) = {a}
getIdentifiers(mk(a, b, c, -, 1) = {a, b, c}
```

Translating the bind list In a VDM++ set bind, patterns are used to indicate which values to select from the set. It is possible to say in a bindList:

```
pattern in set someSet
```

To remove this pattern, the first step is to replace this with the HOL expression:

```
freeVariable IN <| someSet |>
```

in which `freeVariable` is a variable that has not been used anywhere else in the model and `<| someSet |>` is the regular expression translation of the collection. `freeVariable` will also be the variable that the HOL quantifier will range over, so together with the above, `freeVariable` ranges over any value in the translation of `someSet`.

The next step is to make sure that only values that satisfy the pattern are selected:

```
? setToSeq(getIdentifiers(pattern)) . pattern = freeVariable
```

The `setToSeq` function will turn the elements from a given set into a sequence (the order of the elements is irrelevant). The `pattern = freeVariable` part is the equality test over patterns and variables as explained in Section 3.4.6. All together this states that it is possible to find some values for each of the identifiers in the pattern, such that the quantified `freeVariable` is equal to the pattern. When this is the case, the value of `freeVariable` can also be constructed by filling in values in the pattern and the value therefore satisfies the pattern.

Translating the predicate Having selected a variable that ranges over all values from the set that satisfy the pattern, the last step is to make sure the VDM++ predicate can use this free variable. A direct translation is not sufficient, as the `freeVariable` has just been introduced artificially and is not mentioned anywhere in the original predicate. We therefore translate the original VDM++ predicate to the following VDM++ expression:

```
let
  pattern = freeVariable
in
  predicate
```

When translating this let in the regular way, any of the identifiers in the pattern will be assigned the right values.

The overall translation then becomes:

```
<| Q pattern in set someSet & predicate |> =
Qt freeVariable .
    freeVariable IN <| someSet |> /\
    ? setToSeq(getIdentifiers(pattern)) .
      [|| pattern = freeVariable ||] /\
      <| let
           pattern = freeVariable
         in
           predicate                |>
```

In which Q is a VDM++ quantifier (forall or exists) and Qt is this quantifier translation (! and ? respectively). All other parts are from the previous paragraphs.

The above translation is for one binding. When translating a quantifier with multiple bindings in one bindlist, this quantification can be rewritten to two or more nested quantifiers, which each have a single bind in their list. For example

```
forall b1, b2 &
  P
```

is equivalent to

```
forall b1 &
  forall b2 &
    P
```

The translation of the type bind is similar, except for the '`freeVariable IN <| someSet |>`' part. Due to invariants the replacement for this is not directly obvious. It will therefore be topic of Section 3.5.2, when we discuss the invariant strategy for expressions.

3.4.10 Let be such that

The VDM++ expression let be such that looks a lot like a quantifier. The expression

```
let bind be st predicate
in E
```

will select an arbitrary value that satisfies the binds and the predicate and use this in the expression. The result of the statement is whatever the expression comes up with. HOL has a similar statement, which is called a select. It looks like:

```
@ valueConstruction . predicate
```

This will select an arbitrary value in the same way, but instead of using it in an expression, it will return this value as result.

So the translation nearly has the same components as a quantifier translation. We can say it consists of two parts. The first is the translation of the selector, which is exactly similar to the translation of a quantifier (using an `@` instead of the forall or exists), the source of this translation will be called '`selector`' below. The second part is the incorporation of the expression `E`, which looks like:

```
<| let freeVariable = selector
   in let pattern = freeVariable in E |>
```

In which `pattern` is the pattern from `bind` and `freeVariable` is a variable that is not used anywhere else in the model yet.

3.4.11 Set comprehension

Just as the 'Let be such that', the translation of the set comprehension is very similar to the translation of quantifiers. The general outline of a VDM++ set comprehension is as follows:

```
{construction of elements | bindlist & predicate}
```

In HOL this should become an expression of the following format:

```
{construction of elements | predicate}
```

This is nearly equivalent to the kind of translation of the 'let be such that' expression. The difference is that in this case, we do not have a variable list in the resulting expression[4]. In practice this will only make the translation easier. Translation of any of the other components is equivalent.

3.4.12 Map comprehension

The last expression to translate is the map comprehension. The first intention may be to translate this in the same way as the set comprehension (and thereby in the same way as the quantifiers). But this is not directly possible, since HOL does not support the map comprehension as it does with the set comprehension. So instead of writing a HOL map comprehension, we will more or less be using a work-around by rewriting part of the expression to a set comprehension. But let us first take a look at what the map comprehension in VDM++ looks like:

```
{ construction of map elements | bindlist & predicate }
```

In which the `construction of map elements` always looks like:

```
sourceExpression |-> targetExpression
```

[4]The variable list is implicitly determined by all free variables in the expression. The artificial introduction we used in the quantifier expression is not required here.

To translate this to an expression that will be using set comprehensions, we will have to rewrite the mapping to some kind of set. Since mappings can be considered to be sets, several options are available here. In the current translation the choice is to use pairs (products) to encode the mapping into a set. These pairs will be called maplets from now on, as they should be the elements of the resulting mapping. The currently used set comprehension to construct the set is as follows:

```
mset = { mk_(sourceExpression, targetExpression) |
         bindList & predicate }
```

This is not a HOL expression yet, it is still VDM++. As we have seen in previous translations, we are rewriting VDM++ code to VDM++ code here, after which we will use an existing translation to continue the process. In this situation we will use the already defined set comprehension translation to translate the above VDM++ expression to a HOL expression.

The translation makes use of the select (`@`) expression of HOL, we will select an arbitrary value that 'has the right mappings in it'. At first we might want to use the following expression:

```
@ m . (!maplet . maplet IN mset =>
                 FAPPLY m (FST maplet) = (SND maplet))
```

The function `FST` will select the first element from a pair and `SND` the second. Since they are applied to the maplets from `mset`, these correspond to the source value and target value of a single mapping value respectively. The overall expression then states that we want to select the value `m`, for which holds that for each maplet in the above constructed set, when m is applied to the source of the maplet, the result will be the target of the mapping. In other words, the mapping `m` should conform with the set `mset`.

Although this may at first sight seem sufficient, this restriction on `m` is too limited. `m` can have any additional mappings, that were not supposed to be in it. To prevent the result from having too many mappings, the domain of `m` should also be restricted to the desired domain. To get the domain of the mapping from our set, we could select all 'first' values from all the pairs, or simply make a different set. In practice the second is the easiest. The total expression will therefore become:

```
@ m . (FDOM m = sourceMset) /\
      (!maplet . maplet IN mset =>
                 FAPPLY m (FST maplet) = (SND maplet))
```

In which:

```
sourceMset = { sourceExpression | bindList & predicate }
```

The full translation becomes:

$\langle|\{E_s|\text{->}E_t \mid b \;\&\; P\}|\rangle$
$=$
$@m.(\text{FDOM } m = mset_{source}) \wedge (\forall_{e \in mset} \text{FAPPLY } m\ (\text{FST } e) = (\text{SND } e))$

In which:

$$mset = \langle|\{\text{mk\_}(E_s, E_t) \mid b \ \&\ P\}|\rangle$$

and

$$mset_{source} = \langle|\{E_s \mid b \ \&\ P\}|\rangle$$

3.5 Translating functionality

Having discussed the details of translating expressions and types, we now get to the point where we can actually look at translating entire models. Models that consist of independent types and independent expression are virtually useless. It is the collaboration of the individual components of a model that holds one of the keys to quick and easy development. The main means of collaboration is definition. By defining functions or types, they can be used at other parts of the model. They reduce code, ease development and reduce the probability of making errors by preventing having to write code twice.

The discussion of types has already focused on the definition of types, so this will only be discussed briefly here. Definition of functionality (functions) has been left untouched and will be the main topic of this section. The later parts of this section will look at two more topics that are important when dealing with these definitions, namely dependencies and invariants.

3.5.1 Definitions

There are several syntactical differences between definitions and other parts of the model (types and expressions). One of the main difference is that definitions cannot be used as part of anything else. Expressions can be used as part of other expressions: the expression

```
x + 1
```

is part of the following expression:

```
(x + 1) / 2
```

The same holds for types (the type `integer` is part of the type `integer | boolean`). But definitions cannot be part of definitions in VDM++, let alone part of expressions or types. A reference to whatever is defined in the definition may be used, but not the definition itself.

As a result, definitions have to be put at the highest level of a model. And indeed, in a VDM++ model, when definitions are used, they can always be found as the main building blocks of the model. There is no need to search for them, they are always the first to encounter when looking at the top structure of the model. In HOL this distinction of levels is similar, except that the distinction has been made even clearer. As explained in the introduction on HOL, it knows two levels: a meta level with which the HOL engine is controlled and an object level at which the model is expressed and all proofs will take place. All expressions and types will be handled at the object level. From a HOL perspective, these are components of the model that can be used directly throughout

the proof. The definitions on the contrary will at first be expressed through the meta level. Fortunately, when having requested the definition through the meta level, its result will be available at the object level, since its usage in proofs is usually crucial. However, the original request for definition itself will be at the meta level.

As mentioned earlier, there are two types of function definitions in VDM++: explicit and implicit. Both can be handled by the translation, but they have to be translated differently:

Explicit functions A VDM++ explicit function declaration has the following layout:

```
functionName : T1 * T2 * ... * Tn -> Tn+1
functionName (p1, p2, ..., pn) == E
pre Pr
post Po;
```

`T1` to `Tn` are the parameter types. Each of them can be any of the previously discussed types. `Tn+1` is the result type of the function, which can also be any of the discussed types. `p1` to `pn` are patterns that denote the local parameters that will be used to store all or some of the input values of the function. `E` is the expression that indicates how the function should be evaluated. And `Pr` and `Po` are the pre- and postcondition respectively. Both are predicates (expressions resulting in a boolean value).
`E`, `Pr` and `Po` can all use the identifiers declared by `p1` to `pn`. In addition to this, Po can also use a reserved keyword '`RESULT`', that indicates the result of the function, or in other words, the evaluation of `E`.

For example, we could have:

```
sum : (int * int) * (int * int) -> (int * int)
sum ( mk_(x1, y1), mk(x2, y2) ) ==
  mk_(x1 + x2, y1 + y2)
post RESULT.#1 = x1 + x2 and
     RESULT.#2 = y1 + y2;
```

The precondition has been left out. If so, it will automatically be interpreted as true. Also the postcondition may be left out in the same way.

Before the actual translation can start, we have to remove all the patterns in this declaration. They make the translation unnecessarily complex and can be written out relatively easily. Therefore, the above definition is rewritten to the following VDM code:

```
functionName : T1 * T2 * ... * Tn -> Tn+1
functionName (i1, i2, ..., in) ==
  let p1 = i1,
      p2 = i2,
      ...
      pn = in
  in
```

```
      E
pre
  let p1 = i1,
      p2 = i2,
      ...
      pn = in
  in
      Pr
post
  let p1 = i1,
      p2 = i2,
      ...
      pn = in
  in
      Po;
```

`i1` to `in` are simple variables, not patterns. The result may look somewhat more complex due to the extra let expressions, but for these a translation has already been derived. So there is no need to worry about those any longer. We will write E with the above let expression surrounding it as `E'` and do the same for `Pr` and `Po`. Then the result of this first step is:

```
functionName : T1 * T2 * ... * Tn -> Tn+1
functionName (i1, i2, ..., in) == E'
pre Pr'
post Po';
```

Nearly the same definition, but this time without the patterns and some additional let expressions. In our example this would become:

```
sum : (int * int) * (int * int) -> (int * int)
sum (i1, i2) ==
  let mk_(x1, y1) = i1,
      mk_(x2, y2) = i2
  in
      mk_(x1 + x2, y1 + y2)
post
  let mk_(x1, y1) = i1,
      mk_(x2, y2) = i2
  in
      RESULT.#1 = x1 + x2 and
      RESULT.#2 = y1 + y2;
```

The next step involves the translation itself. To define functions in HOL, the translation uses the meta level function '`Define`' defined as part of the HOL system. This function will define any function for us, such that it can be used inside the HOL model. Using this, the basic translation is rather straightforward and will result in:

```
Define `functionName (<|i1|> : <T1>)  (<|i2|> : <T2>) ...
  (<|in|> : <Tn>)  =  <|E|> : <Tn+1>`;
```

The <...> denotes the regular type translation and the <|...|> denotes the regular expression translation. The ':' in HOL denotes a type binding. In the expression above it means that the translation of i1 has to be of type translation of T1, the translation of i2 has to be of type translation of T2, etcetera. Anything on the left of the '=' sign are the parameters of the function and their types (curried, so not separated by commas) and anything on the right of the '=' sign is the function body and its type.

The pre- and postcondition have been left out of the translation. These do not add information to the function itself. They can be used to check the function body or check the usage of the function, but do not change the function definition. Since these checks are part of the proof obligations of the model, the pre- and postcondition may be referred to (there are even likely to be more proof obligations that refer to these pre- and postconditions). We will therefore define them separately, using the name that is being used to reference them: `pre_functionName` and `post_functionName`:

```
Define 'pre_functionName  (<|i1|> : <T1>)  (<|i2|> : <T2>) ...
  (<|in|> : <Tn>)                    =  <|Pr|> : bool';
Define 'post_functionName (<|i1|> : <T1>)  (<|i2|> : <T2>) ...
  (<|in|> : <Tn>)  (RESULT : <Tn>)  =  <|Po|> : bool';
```

The combination of these three statements (meta language statements) involves the entire definition of an explicit VDM++ function in HOL.

Implicit functions Implicit function definitions have a lot in common with the explicit ones. Therefore, we will use the translation of the explicit function definition to translate the implicit one. The actual translation of implicit functions involves one new step, which is more a rewriting then a translation. Just like the first step in translating explicit functions, it will rewrite VDM to VDM.

Let us first take a look at what implicit function definitions look like in VDM++. The general layout is:

```
functionName (p1 : T1, p2 : T2, ... , pn : Tn) r : Tn+1
pre Pr
post Po
```

The first line more or less holds the information that is stored in two lines in the explicit definition. The meaning is as follows: `p1` to `pn` are the patterns used for the input parameters. `T1` to `Tn` are the types of each of these patterns and `r` is a name given to the result of the function. `Tn+1` is the type of the function result. Giving a name to the result of the function is in some cases useful when specifying the postcondition (to improve readability). The name has the same function as the identifier `RESULT`, which may also still be used.

The outline given above is one variation of writing an implicit function definition. There are a lot more variations, but they can all be rewritten to this format in a rather straightforward way and are therefore not relevant here.

To rewrite this definition to an explicit format, it is required to give an exact expression of how to calculate the result. At first sight, that may seem

extremely hard or even infeasible. Fortunately, though, VDM++ holds a useful expression that can solve this problem for us: the let be such that expression. Given a restriction (a predicate) and if desired a type, this expression will find us a value that satisfies the restriction and is member of the type. How to do this is not directly clear (even if it would be possible to do so), but that is usually not relevant when trying to prove anything about the model.

The implicit definition above is rewritten to:

```
functionName : T1 * T2 * ... * Tn -> Tn+1
functionName (p1, p2, ..., pn) ==
  let r : Tn+1 st Po
  in  r
pre Pr
post
  let r = RESULT
  in  Po;
```

All types and parameter patterns are copied and a body of a `let be st` expression is used. The precondition is copied and the postcondition copied, surrounded by a let, to handle the new name given to the result value (`r`).

3.5.2 Invariant strategy

Since the invariants could not be included in the type definitions, they have to be reintroduced into the model. Whenever a type is being used, in some way its invariant (if it has one) has to be stated, such that the semantics of the models (VDM and HOL) are equivalent. Since the number of places in which types can be used are limited, we will discuss each of them and show how to reintroduce the invariant.

Including the invariants explicitly is a sub step of the translation. It is therefore not a VDM to HOL translation, but a translation from one intermediate format to another. Since the implementation is not directly relevant, this section will only focus on the meaning and especially the semantical equivalence of the sub translation. Since the intermediary formats do not clarify this, mainly logical formulas are used to indicate the translation.

In each of the formulas below, we assume a definition of the types T, T_1 and T_2. The invariants $invT$, $invT_1$ and $invT_2$ have been imposed on them respectively and their super types are T', T_1', T_2' respectively. The definition of T would in VDM look like:

```
T = T'
inv x = invT(x)
```

In a logical formula this is:

$$\forall_x.x : T \Leftrightarrow (x : T' \wedge invT(x))$$

Stating that all values x are of type T if and only if they are of the super type T' and the invariant of T holds.

Quantifiers The original expression, in which the type bind occurs would be:

$$\forall_{x:T}.P(x)$$

For some predicate P. This can be rewritten to:

$$\forall_{x:T'}.invT(x) \Rightarrow P(x)$$

Since we only want the predicate to hold for all values that are of type T that satisfy the invariant.

The existence quantifier is slightly different:

$$\exists_{x:T}.P(x)$$

Goes to:

$$\exists_{x:T'}.invT(x) \wedge P(x)$$

Since we are searching for a value that does not only satisfy the predicate, but that also is of type T and therefore satisfies the invariant.

Other quantifier-like translations Equivalent to the universal quantifier, there are several more translations that simply introduce the invariant by means of a conjunction. These are the set comprehension, map comprehension and let be such that expression. Their translations are given below:

- $\{E \mid x : T \ \& \ P(x)\}$translates to $\{E \mid x : T' \ \& \ invT(x) \wedge P(x)\}$
- The translation of the map comprehension is the same as the set comprehension.
- $let\ x : T\ st\ P(x)\ in\ E$ translates to $let\ x : T'\ st\ P(x) \wedge invT(x)\ in\ E$

Function definition Translating function definitions involves two or more invariants, since they can carry more than one type. The source of the translation would be:

$$\begin{aligned} &f : T_1 \rightarrow T_2 \\ &f(x) == E(x) \\ &pre\ Pre \\ &post\ Post \end{aligned}$$

In which Pre and $Post$ are predicates.

Translating this would need to include both the source and the target invariants. This is done as follows:

$$\begin{aligned} &f : T_1' \rightarrow T_2' \\ &f(x) == E(x) \\ &pre\ Pre \wedge invT_1(x) \\ &post\ Post \wedge invT_2(RESULT) \end{aligned}$$

The variable $RESULT$ used in the postcondition has the same meaning as it always has in postcondition. It holds the result of the function.

Note that calling the function f on an arbitrary value v generates a proof obligation that v satisfies the precondition of f. Although the precondition above has been changed, this does not change the proof obligation, since the above translation is performed after the proof obligations were generated. The code above should therefore explicitly not be seen as another VDM model that can be used as input to a translation. The same can be said about the post-condition and the proof obligation that is generated to ensure that any result generated by the function satisfies this condition.

More parameters can easily be added by adding the invariants of their types as extra conjunctions to the pre and postconditions in the same way as is done above.

Type definition The last location where types can be used are in type definitions themselves. We distinguish two kinds of type definitions. The first is the regular type definition of a new type which will be called T_{new}:

$$T_{new} = T$$
$$inv\ x = invT_{new}$$

This will be translated to:

$$T_{new} = T'$$
$$inv\ x = invT_{new} \wedge invT(x)$$

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:

$$T_{new} = \ldots f_i : T \ldots$$
$$inv\ x = invT_{new}$$

This will become:

$$T_{new} = \ldots f_i : T' \ldots$$
$$inv\ x = invT_{new} \wedge invT(x.f_i)$$

In both constructions and practically all equations of this section, the $invT$ is somewhat special, as it is not the invariant of type T defined in the model, but the $invT$ of the result of the translation above. This may sound somewhat cryptic, but the reason can be found in the two type translations above. Whenever a subtype is made, the invariant is changed by including the invariants of the super type (or the types of all fields in case of a record). This is not only applied one step, but recursively. It is this recursively constructed invariant that should be used.

The second of the above translation could also be applied to record types with multiple fields of non-primitive types. Instead of using one field invariant in the conjunction above, a longer, similar conjunction can be made containing multiple field invariants.

Note that when constructing a value of the new type (T_{new}), a proof obligation will be generated. This proof obligation states that the value satisfies all invariants on it (of its own type as well as of all super types). Yet, at the time of writing, this invariant is not always constructed correctly in the VDMTools set. At least in the case of a regular subtype (the first of the two kinds above), the super invariants are left out and never checked (the bug has been reported).

3.6 Model dependencies

One topic, that is currently handled in translation has not been addressed yet: dependencies within the model. Function A might refer to a function B by means of a call. In VDM++, the order of declaration is meaningless. In HOL, anything has to be defined before using it. Therefore, function B must be defined before function A can refer to it. To solve this dependency problem, all dependencies are determined during translation and the definitions are sorted in a way that all dependencies are solved: the first few definitions typically have no dependencies, then a series of definitions may be inserted that only use this first set of definitions and so on.

This sorting will do fine, until we reach a point where definition A depends on definition B and definition B depends on definition A. In this case of mutual recursive dependencies, none of the definitions can be defined first. We need a mechanism like forward declarations as used in the language C, to solve such a problem. As mentioned earlier, HOL does not directly support this. Mutual recursive dependencies are therefore not allowed as input to the translation.

In addition to an error in a mutual dependency situation, the code will also produce an error when a dependency is not met. This should however already have been noticed by the checks on the VDM++ model.

Chapter 4

Automated proof support

Being able to translate a VDM model, we can now take a look at proving obligations about the model. This chapter will start by giving a more detailed description of proof obligations, tactics and proof in HOL. The rest of the chapter is devoted purely to proof of the obligations. Different sections are devoted to the different categories of proof obligation introduced below.

As mentioned earlier, this is the part where the proof takes place, but that does not make it the only part where the proof can be influenced. A great deal of influence on the proof is already embedded in the previous chapter by using a model translation suitable for proof. The text below will explain the extension of the proof process that has its basis in the translation.

4.1 VDM++ Proof obligations

VDM++ proof obligations are predicates using the definitions in a model they were generated from. Each obligation corresponds to one or more possible inconsistencies in its model. Their validity implies the absence of these inconsistencies.

The proof obligation generator in the VDMTools toolbox distinguishes 51 types of proof obligations for VDM++ models. Dealing with each type in a different way would be rather complex and time-consuming in development. Furthermore, many of the types can be proved in similar ways and do not require their own mechanisms. Therefore, a categorization will be followed that was earlier proposed and used in [13] as part of the PROSPER project. The following groups of proof obligations can be distinguished:

- *Domain checking*: This holds the obligations resulting from the use of partial functions and partial operators. These proof obligations are mainly caused by the use of preconditions on functions or on operators.
- *Subtype checking*: Proof obligations that are required due to the use of subtypes. In particular due to the use of invariants.
- *Satisfiability of implicit definitions*: Proof obligations that are the result of the use of implicit function definition. For each of these, a proof has to be given that the implicitly defined function can have a result for any input.

- *Termination*: For each recursive function a proof is required that it will always terminate.

For each of these categories, a section of this chapter explains what the obligations look like, how they can be proved and in what way they are handled in the current implementation.

4.2 Tactics

Tactics are the most important tool to guide proof attempts in automated proof in HOL. The first part of this section will focus on the context, meaning and usage of tactics in HOL. The second part will focus on how to construct HOL tactics.

4.2.1 The context of HOL tactics

Theorems in HOL are predicates. The only way to construct theorems is by applying rules of inference to axioms, or by applying rules of inference to existing theorems. The sequence of such application corresponds to a 'logician's proof'.

There are five primitive axioms in HOL and eight primitive inference rules. An inference rule is a function that results in a theorem. The only way to construct a new inference rule is by using axioms or existing inference rules. This way, the construction of any theorem can be traced back to the axioms and primitive inference rules in HOL and is thereby proved[1].

The general format of a theorem is: $t_1, \ldots, t_n \vdash t$. t_1 to t_n are predicates indicating the assumptions and t is a predicate indicating the conclusion. A theorem states that if its assumptions are true, so is its conclusion. A theorem might also have no assumptions at all, which is denoted by $\vdash t$.

A proof attempt in HOL is an attempt to construct a theorem. Once the way to construct the theorem is known, so is its proof. But before we have this theorem, we need a way to express what we are trying to prove. This is what a 'goal' does. Goals are the main ingredients of a proof attempt in HOL. A goal is a pair $([t_1; \ldots; t_n], t)$, that states what we want to prove: t_1 to t_n is the list of assumptions and t is the term we want to prove. In other words, the goal $([t_1; \ldots; t_n], t)$ states that we are trying to find a proof of the theorem $t_1, \ldots, t_n \vdash t$

Finding a proof is basically a backward search. We start at the goal and try to construct the proof by 'decomposing' the goal. Exactly in the opposite way that the theorem we are looking for would be constructed. Inference rules in HOL do not work backwards, they are just functions and can only be used in one way. Inversion is not directly an option. Furthermore, it is not easy to tell which inference rules should be used. Since the construction of a single theorem might require applying the primitive inference rules thousands of times, we will have to search in an efficient way. This is where tactics come into focus.

Tactics are functions that can be applied to a goal and result in a list of sub goals and a justification function. This justification function is an inference rule that can take the achievements of the sub goals as input and result in the

[1] At least it is proved, assuming that the axioms and primitive inference rules are sound. As long as no axioms are added, this seems to be a 'fair' assumption.

achievement of the main goal. Or in other words, once theorems have been constructed corresponding to the sub goals, this justification function will construct a theorem corresponding to the main goal (by means of inference, since this is the only way to construct theorems).

An example of a HOL tactic is `CONJ_TAC`. Suppose our goal is to prove $\vdash A \wedge B$ (meaning that our HOL goal is $([], A \wedge B)$), then CONJ_TAC will split this goal into the sub goals $\vdash A$ and $\vdash B$. The inference rule given by this tactic, which proves the main goal from the sub goals, is the $\wedge$-introductioninference rule which corresponds to the logical rule:

$$\frac{\Gamma_1 \vdash t_1 \qquad \Gamma_2 \vdash t_2}{\Gamma_1 \bigcup \Gamma_2 \vdash t_1 \wedge t_2}$$

4.2.2 Constructing tactics

Since tactics are ML functions, they can be constructed by the regular function definition of the ML language. A tactic can consist of trying to use a single rule of inference and possibly fail during the process (like the `CONJ_TAC` tactic that we have seen in the previous section), but a tactic can also consist of a fully automated backtrack over a large base of inference rules. Either way, they are constructed as regular ML functions.

To aid in the process of constructing tactics, there are some building blocks for constructing tactics. These building blocks are called 'tacticals'. A tactical is a function itself, that takes one or more tactics as its input and produces a new tactic as output. The most frequently used example of a HOL tactical is the function `THEN`. Assuming `T1` and `T2` both are tactics, then the following expression will give us a new tactic:

```
val T = T1 THEN T2
```

The tactic `T` resulting from this expression will first try to apply `T1` to the goal. If `T1` fails, so will `T`. If `T1` succeeds, `T` will try to apply `T2` to the sub goal that `T1` has come up with. Again, if `T2` fails, so will `T`. If `T2` succeeds, T will succeed too and result in whatever the sub goal was that `T2` came up with. In other words, the overall tactic will try the proof that can be found by sequential application of `T1` and `T2`.

Another frequently used tactical is the ORELSE:

```
val T = T1 ORELSE T2
```

The tactic `T` that will result from this expression will first try `T1`, if this succeeds, so will `T`, if it fails, it will try `T2`. If `T2` succeeds, so will `T`, otherwise `T` will fail. Or in other words, `T` will try to apply `T1` first and if unsuccessful, it will try `T2`.

HOL holds about 15 predefined tacticals, including a `REPEAT` tactical and several to deal with more then one sub goal. Even though tactics can still be written as straightforward functions (including function headers, failure tests and types), using tacticals significantly speeds up the development process. Unfortunately the tacticals are not always semantically sufficient, which was the case once or twice in the current implementation of tactics. However, in this document we will only construct tactics via tacticals and will not look at

construction using function definitions[2].

To use tacticals, we need tactics. We can use any tactic we have defined ourselves as well as any ready-made tactic. Even though at first sight, customly defined tactics may seem more focused on the task at hand, we will see that also the predefined tactics can be customized to a great extend. At a higher level of abstraction, we can distinguish two kinds of predefined tactics:

- Decision tactics
- Rewriting tactics

The first set is more or less what one expects from a tactic. It will use a set of inference rules to search for a proof, usually using a variation of a backtracking algorithm. The set of rules used can in most cases partially be determined by the user. The main difference between tactics within this category is their search strategy. Some are very weak (in that they will not find many proofs), but are relatively fast, others are the other way around. In general, the performance of these tactics decreases rapidly when adding more inference rules. Simply adding an entire theory is hardly ever an option, this would slow down the tactic far too much. Adding carefully selected rules is the way to keep these tactics useful. Examples of decision tactics in HOL are: `DECIDE_TAC`, `MESON_TAC` and `EVAL_TAC`.

The second set of tactics are the rewriting tactics. HOL has a strong rewriting subsystem. Any theorem can be converted to one or more rewriting rules and many rewriting rules are already available. In contrast to the decision tactics, the rewriting system is based on an ordering of terms. A rewrite is only applied when it will decrease the term's rank on this ordering. Usage of this ordering ensures termination and prevents many useless rewrites (e.g. rewriting in loops, or unlimited expansion of a term). In contrast to the provers, rewriters are fast and will always terminate. Furthermore, the decrease of their computational performance due to added rewrites is insignificant. The disadvantage of this rewriting system is that due to the ordering, not all proofs can be found. But we will see that in practice, rewrites are still widely applicable and rather strong. Examples of rewriting tactics are: `REWRITE_TAC`, `SIMP_TAC` and `RW_TAC`.

Both decision tactics and rewriting tactics are 'loaded' with a default amount of inference rules or rewrite rules. In addition to these, the user can add extra rules to increase the search or rewrite space. In most situations, expansion of these sets is done by using existing rules, but in some situations new rules are required. If that is the case, a new theorem has to be constructed. This theorem may both be turned into rewrite rules as well as into inference rules. Proof of the theorem is by its construction. We will therefore also see construction (and proofs) of theorems that have appeared to be useful additions to the theorems already available in HOL.

[2]Although we do not look at function definitions as such, this does not mean that the tactics we use are not functions. A tactical simply yields a function as a result and one could say that that makes the tactical a function constructor itself, just as the regular function definition is.

4.3 Domain checking

The first category of proof obligations is the 'domain checking' category. It is in general the most common category and usually also the easiest to prove. This category covers proof obligations generated as a result of the use of partial operators and partial functions. Take for example the following VDM++ function:

```
factorial int -> int
factorial (x) ==
  if   x = 0 then 1
  else x * factorial(x - 1)
pre x >= 0;
```

Any usage of this function, including the usage within the function body (the recursive call) will generate a proof obligation. For example, the following usage:

```
factorialIncr int -> int
factorialIncr (x) ==
  factorial(x + 1)
pre x >= -1;
```

will generate the proof obligation:

$$\forall_{x:int}.\text{pre\_factorialIncr}(x) \Rightarrow \text{pre\_factorial}(x+1)$$

Which can be rewritten to:

$$\forall_{x:int}.x \geq -1 \Rightarrow (x+1) \geq 0$$

Although this rewriting may seem rather obvious when written down in this way, in general it is not. The rewriting will have to be done by the tactic and in such a way that the proof may be found. In some situations (like this one) it is useful to rewrite all definitions, but in others it might be more useful to use earlier proved theorems about higher level functions (such as about the `factorial` or `factorialInc` functions here).

From inspection of the domain checking proof obligations in the case studies, it can be determined that most of them consist of relatively simple basic logic. Even though the example above already contains some simple arithmetics, in general, many of the domain obligations can be proved by using some simple logical inference rules. An attempt to prove these obligations therefore consists of the usage of several decision tactics of HOL. Each of them is explained below:

TAUT_TAC TAUT_TAC is a tautology checker. Given a valid goal, it will prove that it is valid by performing Boolean case analysis on the variables of the goal. The goal used as input has to be a propositional formula, which is a term containing only Boolean constants, Boolean-valued variables, Boolean equalities, implications, conjunctions, disjunctions, negations and Boolean-valued conditionals. Universal quantifications over the variables in the goal are allowed. The tactic will fail if the goal is not valid, or the input is not a propositional formula.

The tautology checker has been made for rather simple formulas. There are very few proof obligations in regular VDM++ models that satisfy the criteria of being propositional. Most of them either contain existential quantifiers, or the usage of functions or operators that are not directly Boolean related. Therefore, the tactic will usually fail without even trying a proof. If however such a simple goal does occur, it will do its job relatively quickly.

MESON_TAC MESON_TAC is a decision procedure for first order logic. It uses a PTTP (Prolog Technology Theorem Prover) based implementation of model elimination (see [32] for details). The tactic will fail if the input is not a first order predicate, the input is not valid, or if the maximum search depth has been exceeded during the proof attempts.

The MESON_TAC tactic can take a list of theorems as its argument that can be used in its search. The choice of this list is therefore crucial to its success. This choice is also of direct influence to the performance of the tactic. A long list will give a high branching factor in search and therefore increases the time to find the proof or reach the maximum search depth. An empty list may also be provided. Only basic predicate calculus will then be used. Due to the additional theorems, this tactic is frequently the slowest of the tactics discussed here. However, when given an empty list, its speed is comparable to most other decision tactics.

In the current implementation a slightly different version of the MESON_TAC is used: GEN_MESON_TAC. This tactic gives more control over the search space. In contrast to the MESON_TAC, GEN_MESON_TAC includes the assumptions of the goal as additional information in its search, furthermore, it has some additional parameters:

```
GEN_MESON_TAC min max step thms
```

`min` is the search depth to start at, `max` is the maximum search depth, `step` is the number with which the search depth is incremented and `thms` is a list of additional theorems that can also be given to MESON_TAC. Current implementation uses the following customization for the domain obligations:

```
GEN_MESON_TAC 0 10 1 []
```

DECIDE_TAC DECIDE_TAC attempts to prove M using a propositional tautology checker and a linear arithmetic decision procedure. This tactic does not have many restrictions on the format of the input, except that it should be a predicate. The tactic fails if the proof attempt fails or the input is incorrect. The tactic does not take much time, but is also not very powerful as most of its original functionality has been taken over by the arithmetic simplification set which will be discussed later. The original cooperation of decision procedures it used to incorporate in earlier versions of HOL is no longer useful now and has been removed. The cooperating decision procedures might however be reinserted in a different format in a new version of HOL, at which point this tactic will gain much more strength and will become much more useful than in the current implementation. The reason to include it now is that in rare situations it might give useful results and it will probably be more powerful in a new version of HOL. A strength that might be useful when trying to prove obligations that

currently fail to be proved automatically. Either way, in the current version it will use hardly any execution time.

VDM_ARITH_TAC VDM_ARITH_TAC is the arithmetic decision tactic used in the current implementation. It is not a built-in HOL tactic (hence the name containing VDM), but has been defined in this project. It consists of three tactics, namely:

1. numLib.ARITH_TAC
2. intLib.ARITH_TAC
3. realLib.REAL_ARITH_TAC

Each of them hold a decision procedure based on various decision procedures for arithmetics (such as one based on Presburger arithmetic [29] and one on the Omega test algorithm [31]). They are all called in sequence until one of them succeeds in proving the goal. The HOL definition of the tactic is:

```
val VDM_ARITH_TAC =
(
  numLib.ARITH_TAC                    ORELSE
  intLib.ARITH_TAC                    ORELSE
  realLib.REAL_ARITH_TAC
);
```

REDUCE_TAC The last tactic called for domain checking proof obligations is the REDUCE_TAC. This tactic uses deductive steps to perform any arithmetic or Boolean reductions possible. It will try to achieve true or false along the way. Whatever the result of the reduction, the tactic will never fail. The result from the reduction will be returned as sub goal. The input is restricted to all standard Boolean and numerical operators. If supplied with incorrect input, the tactic will not perform any reductions, but will always succeed anyway.

Combining the decision tactics All decision tactics discussed above are combined into the VDM_GENERIC_PROVE_TAC tactic as follows:

```
val VDM_GENERIC_PROVE_TAC =
(
  tautLib.TAUT_TAC                    ORELSE
  (mesonLib.GEN_MESON_TAC 0 10 1 [])  ORELSE
  DECIDE_TAC                          ORELSE
  VDM_ARITH_TAC                       ORELSE
  reduceLib.REDUCE_TAC
);
```

Note that REDUCE_TAC is deliberately placed at the last position. This tactic will always succeed, which has as result that due to the use of the `ORELSE` tactical the VDM_GENERIC_PROVE_TAC also always succeeds. Even though VDM_GENERIC_PROVE_TAC may not be able to prove the goal, it will

still give it his best shot by trying to reduce it for the user.

Many domain checking proof obligations can be proved using the tactic above, but not all of them. Some, for example the one given at the beginning of this section, require rewriting of definition, which is slightly more complex than simple logical reasoning. Others might also require a cooperation between rewriting and decision procedures. That is why the tactic above is not the only one tried. In the next section we will look at a more sophisticated version of the tactic when dealing with subtype checking. This more sophisticated tactic will be tried if the above fails. All of the domain checking proof obligations can be proved for the case studies considered (when using both tactics). We will see examples of this in the next chapter.

4.4 Subtype checking

The next category of proof obligations to be discussed is the one resulting from subtypes. These proof obligations are next most common after the domain checking obligations. They mainly result from the use of invariants. Take for example the `specialNat` type from Section 3.3.1:

```
types:
  specialNat = nat
  inv x == x <> 2
functions:
  sum : specialNat * specialNat -> specialNat
  sum (x, y) ==
    x + y;
```

In the sum function, a new value is constructed by adding the two parameters. This raises the questions whether this new value actually satisfies the functions signature and is indeed a `specialNat` value. Due to the signature of the + operation (`nat * nat -> nat`), the value is certainly of type `nat`, but the invariant does not necessarily have to be satisfied. To verify this, a proof obligation is generated:

$$\forall_{x:\text{specialNat},y:\text{specialNat}} \cdot \text{inv\_specialNat}(x + y)$$

Which can be rewritten to:

$$\forall_{x:\text{specialNat},y:\text{specialNat}} \cdot (x + y) \neq 2$$

Which does not hold and can therefore not be proved for this model. This might for example be solved by adding a precondition to sum.

Although the example above may be relatively simple, proving invariants to be satisfied is usually rather complex. The 'core' of the proof usually consists of a reasonable amount of inference rules applications. But getting to the core requires working through a maze of definitions, simplifications and rewrites. Furthermore, the proof obligations themselves usually become rather long, making the work even more complex, tedious and especially error-prone in the case of manual proof. Since computers do not have too much trouble with

keeping track of large formulas, or doing many steps before getting to the real proof, automation of the proof of this category of proof obligations is very useful.

Clearly, we could try the tactic used on the domain checking proof obligations, but in practice this will not give too many results (in most case studies no additional proofs at all). This tactic is therefore extended using simplification and more proof support. The following subsections will first focus on rewriting and additional decision support individually and then focus on using both at the same time when looking at the complete subtype checking tactic.

4.4.1 Simplification support

As mentioned above, the proof obligations in this category require much rewriting before getting to the core of the proof. The rewriting is basically part of the regular proof and can therefore be done by a decision tactic if desired. In general, however, rewriting tactics are preferred because of their high speed and ability to handle many more kinds of rewrites. The main tactic used in this thesis has a separate part to handle the rewriting: `VDM_SIMPLIFICATION_TAC`. This tactic is constructed based on experience with the test cases discussed in the next chapter. Whenever additional rewrites were required, the tactic was extended. It consists of sequential application of several 'sub'-tactics to the goal. Each of them will be discussed below. To make the explanation somewhat easier, they are not explained in the order they are used in the tactic. Their order will be discussed later.

Stateful simplification Most of the tactics mentioned so far make use of additional parameters that can widen their search or rewrite space. In the case of decision tactics these are usually provided in the format of a list of theorems. In the case of rewriting tactics these are usually in the format of a simplification set. This simplification set is a set of rules along which hopefully the theorem can be simplified. A simplification set can, with some calculation, be constructed from a set of theorems.

One rather special set of rewriting tactics does not use this additional parameter, but the set of simplification rules used can still be extended. These tactics make use of the so-called 'stateful' or 'implicit' simplification set called `srw_ss`. This set of simplification rules is maintained in the background of HOL and can be requested at any desired time. The set is filled in one of three ways:

- Any types that are defined in the HOL system are not only stored in the HOL type base, but also in the `srw_ss` set. All definitions and most basic theories about the types are rewritten to simplification rules and stored directly at the time of definition. This allows easy and direct access to all type related theories.

- A (predefined) theory that is loaded into the HOL system can add any of their simplification rules to the simplification set. What rules are added is decided upon by the developer of the theory, hoping that his selection will be useful in most situations. In practice this method of augmenting the simplification set will provide it with most of its content. Furthermore, it is the easiest way of incrementing the set, as it only requires one load

command of the theory to expand it with all simplifications on a certain topic.

- The last method is by direct user interaction. Whenever a definition has been made, or a theorem proved, their simplifications may be extracted and added to the simplification set.

Each of these three methods has been used extensively in this project. However, only the last will be clearly visible in the result of the model translation. It involves additional statements after each definition, that will explicitly add the definition's simplification rules to the set. Although we have ignored these statements in the description of the translation of the model, the current implementation will automatically add them whenever a definition is made (which is basically after every ML-statement of the translated model).

Before looking at the tactics that use this simplification set, it is important to realize that the set cannot directly be referenced as a constant. It can change at any moment in time and therefore has to be generated by a function which is also called `srw_ss`. This may at first sight seem to be rather straight forward: instead of using a constant, just call the function. But in practice, this unfortunately will not always work. Let us look at a usage of the `ASM_SIMP_TAC` rewriting tactic (ignoring its meaning for now):

```
val SRW_ASM_SIMP_TAC = ASM_SIMP_TAC srw_ss() [];
```

This piece of code will apply the `ASM_SIMP_TAC` to the simplification set that has automatically been constructed in the background and to an empty list, which is just another parameter and irrelevant here. The result will be a tactic that will use stateful simplification set in the 'ASM_SIMP way'. The result is stored in the `SRW_ASM_SIMP_TAC` value. This seems to be a perfect definition of new tactic that we can use to simplify a goal. When we will use the tactic a few times, this will even seem to be true. It will indeed use the simplification set and it will indeed simplify our goal. But what we do not directly see, is that we are constantly using an old version of the simplification set. The `srw_ss` function is only evaluated once: at the time of the definition. Regardless of any increased power of the real simplification set, this tactic will stay the way it is and not use any of it.

To solve this, we need to reconstruct the set every time the tactic is used. The only way to do this is to write down the definition above as a function definition. Thereby, `SRW_ASM_SIMP_TAC` will no longer be a tactic, but a function resulting in a tactic. It can be evaluated any time it is required. The definition of the function itself is not directly relevant now, but the concept is as it will be used throughout the tactic.

When used in the right way, the stateful simplification set can be a very strong tool. It especially removes the burden of keeping track of all available definitions and theorems. Furthermore, it eases the usage of simplification sets of existing theories. In our tactic, we use the simplification set in two custom tactics: `SRW_RW_TAC` and `SRW_FULL_SIMP_TAC`.

In the `SRW_RW_TAC` tactic, `srw_ss` is used as a parameter to the `RW_TAC`. `The RW_TAC` is one of the strongest rewriting tactics. It provides conditional

and contextual simplification, using the current hypotheses when simplifying the goal. In addition to the simplification set, the `RW_TAC` can also take a list of theorems that will be added to the simplification set before using it. These additional theorems can be given as a parameter to `SRW_RW_TAC`. Using this augmented simplification set, it automatically performs case-splitting on conditional expressions in the goal, breaks down any conjunctions, disjunctions, double negations, or existentials occurring as hypotheses. It keeps the goal in 'stripped' format so that the resulting goal will not be an implication or universally quantified. A more extensive description of these and more characteristics of `RW_TAC` can be found in [28] on page 720.

The `SRW_FULL_SIMP_TAC` will use `srw_ss` as a parameter to the `FULL_SIMP_TAC` tactic. This tactic will also take an additional list of theorems, with which it will construct an augmented simplifications set. In contrast to the `RW_TAC`, this tactic will also simplify the hypotheses of a goal. Furthermore, it will also use the hypotheses to construct new simplification rules and even use the results from simplifications to construct new simplification rules. It will work through the simplification of the goal in a 'left to right' way. Although the choice of which simplification to apply next is somewhat different from the `RW_TAC`, it has many of the same characteristics (such as breaking down expressions and case-splitting). A more extensive description of the tactic can be found in [28] on page 305.

We will look at the usage of these two tactics later. Their exact definitions can be found in Appendix D.

Additional theorems The `srw_ss` set is not always sufficient. For certain proofs, additional rewrite rules, or additional inference rules are required. Therefore, the subtype checking tactic holds a list of theorems that can be used in addition to the ones already available. This list is called `tacticSimplificationTheorems` and is used in the following expressions:

```
RW_TAC (std_ss) tacticSimplificationTheorems
SRW_FULL_SIMP_TAC tacticSimplificationTheorems
SRW_RW_TAC tacticSimplificationTheorems
```

Each of these expressions will give us a tactic. The tactics themselves have already been discussed above. `std_ss` is a pre-defined simplification set containing some basic rewrites (it is the most basic, non-empty, simplification set HOL has predefined).

The exact definition of `tacticSimplificationTheorems` can be found in Appendix D.

Let expressions One last tactic used for simplification has not been discussed yet, namely:

```
(FULL_SIMP_TAC std_ss [boolTheory.LET_THM])
```

This tactic is added for the simple reason that HOL does not always rewrite the LET expressions. In some cases all previously mentioned tactics will keep a let expression in a goal, unable to reason about the local variable defined by it. This might be a bug in the HOL tactics as well as a deliberate choice. Either way, it is solved by explicitly including the `LET_THM` theorem at the right location in the subtype checking tactic.

4.4.2 Decision support

In addition to just simplification support, the new tactic to handle the subtype checking proof obligations will also require more decision support. The original tactic only supported proof using inference rules for simple Boolean logic and arithmetics. This will no longer be sufficient when trying to prove subtype checking proof obligations. In these obligations, we will also need additional theories about for example maps and sets. The difficulty is in selecting as few theorems as possible to be able to prove as many proof obligations as possible, since every extra theorem will slow down a decision tactic significantly. The best set depends on the case study, but the the current implementation tries to select a set that is usually sufficient. The theorems currently selected are based on the case studies, but during selection a criteria of generality was more important. They can be divided into two groups:

Pre-defined theorems These contain theorems selected from the HOL libraries. There are currently 14 of these theorems. They mainly deal with maps and sets, as these are usually the types that are subject to invariants (at least this was the case in the case studies). All selected theorems can be found in Appendix D. They are stored in the list that is held by the `tacticBuiltInTheorems` constant.

Custom theorems Although the inference rules available in the HOL libraries are sufficient to prove a theorem[3], they are not always sufficient to prove theorems fast enough. Clearly, just the primitive inference rules and axioms would be sufficient, but the additional libraries provide (among other advantages) extra speed in computation. When, however, these libraries are not sufficient, new theorems can be added. This is also the case in the current tactic. Five theorems have been (carefully) selected, that expand the applicability of the tactic when it has to be run within reasonable amount of time (a minutes or less). The theorems, as well as their proofs can be found in Appendix C.

The additional theorems are used in just one way in the decision support, which is by means of the `PROVE_TAC` tactic. This tactic is an indirect call to the previously discussed `MESON_TAC` tactic. It will use the default options of the MESON library along with some processing of the goal prior to the actual proof, which will not be discussed more extensively since it is not directly relevant here. More information on this topic can be found in [28]. The `MESON_TAC` tactic has been chosen since it is basically the only pre-defined decision tactic in HOL that can handle additional theorems. Furthermore, it is the most commonly used and most powerful tactic when proving goals using decision support. The call to the `PROVE_TAC` looks like:

```
PROVE_TAC (tacticBuiltInTheorems @ tacticCustomTheorems)
```

The list `tacticBuiltInTheorems` constant holds the pre-defined theorems, the list `tacticCustomTheorems` the custom theorems and the `@` operator denotes a list concatenation. Running this tactic might take a long time if the goal is

[3] Meaning that if a theorem can be proved in HOL, then it can be proved with the theorems currently available in the libraries.

complex.

Since the current list of theorems (custom and pre-defined) is based upon the case studies used, it is most likely not sufficient in general. Additional case studies may therefore be advisable to extend the list. The current computational complexity allows expansion.

Note that the order of the theorems in Appendix D may seem somewhat unorganized. This ordering is however relevant when looking at the computational complexity of the tactic. The tactic will expand its search by trying the first theorems in the list first and the later ones later (within a single level of the search tree). It therefore influences the branching behavior of the tactic. The theorems are organized according to their likeliness to be successful in giving a proof (or part of a proof). This ordering is again based upon the case studies used and may not be correct in general.

4.4.3 Combining simplification and decision support

The last step is to combine the two ways of supporting the proof (simplification and decision). They will be combined in one main tactic, just like the one for domain checking proof obligations. This one is, however, slightly more complex. It looks like:

```
VDM_SIMPLIFICATION_TAC THEN
(
  (
    Q.UNABBREV_ALL_TAC THEN
    VDM_SIMPLIFICATION_TAC THEN
    VDM_GENERIC_PROVE_TAC THEN
    (PROVE_TAC (tacticBuiltInTheorems @ tacticCustomTheorems))
  )
  ORELSE
  (
    VDM_GENERIC_PROVE_TAC THEN
    (PROVE_TAC (tacticBuiltInTheorems @ tacticCustomTheorems))
  )
)
```

The tactic will first apply the simplification tactic to the goal. In some situations this alone will prove the goal. Then there are two options:

The first option will try the following steps sequentially:

1. `Q.UNABBREV_ALL_TAC` During rewriting goals, HOL introduces many abbreviations. The definition of abbreviations are stated as hypotheses of the goal and the abbreviations themselves are used in the conclusion of the goal or in other hypotheses. They can be handled rather well during rewrites, but in the case of decision tactics they may prevent them from finding a proof. Removing abbreviations is easy, just substitute the abbreviation with its full description. There is a tactic in HOL that does this for all abbreviations in a goal, namely the `Q.UNABBREV_ALL_TAC` tactic.

2. `VDM_SIMPLIFICATION_TAC` After removing all abbreviations, new rewrites are usually possible. These rewrites were technically already possible in the version with the abbreviations, but they wouldn't have simplified the goal (just made it more complex) and were therefore not used. Since the unabbreviation above has made the goal significantly more complex, new abbreviations may be possible now.

3. `VDM_GENERIC_PROVE_TAC` Having applied all rewriting tactics, hopefully only the 'core' of the goal is left. The first proof attempt is by trying the tactic that was already derived for the domain checking proof obligations. Note that from the definition we have given in the previous section, we know that this tactic will always succeed, if it is not able to prove the goal completely, it will try to make it as simple as possible.

4. `PROVE_TAC` The last step is to use the decision support derived in the previous section. The selected pre-defined and custom theorems will try to construct the remaining proof. This might fail.

In steps 1 and 2, the goal is unabbreviated and simplified. Although the goal will semantically be equivalent, it is rewritten to a different format. This has as a result that some inference rules that might have been used in the original version are no longer available in the version after abbreviation and simplification. In general one could say that the original version allowed for 'higher level' theorems in its proof, while the new version only allows for 'lower level' theorems in its proof[4]. The exclusion of these 'higher-level' theorems can prevent proofs from being found. As an example, let us look at the following goal:

$$([\text{ABB } x = A \to B,\ \neg A \vee x],\ A \to x)$$

It has two hypotheses: an abbreviation x and a regular formula using this abbreviation. The conclusion also uses this abbreviation. Now suppose we also have the following 'higher-level' inference rule:

$$\frac{\neg A \vee B}{A \to B}$$

Using this rule we might directly rewrite the goal to the following sub goal:

$$([\text{ABB } x = A \to B,\ \neg A \vee x],\ \neg A \vee x)$$

Which can in one more step be proved to be true, since the conclusion equals one of the hypotheses. However, if we had rewritten the abbreviation first, we would have got:

$$([\neg A \vee (A \to B)],\ A \to A \to B$$

Which could then have been rewritten to:

$$([\neg A \vee (A \to B)],\ A \to B$$

Although this can be proved too, it will require more than two steps, since there is simply no inference rule that can deal with this goal directly. In this

[4] Higher level and lower level theorems may be defined using the number of steps of primitive inference rules their shortest proofs require.

example the few steps extra may not be relevant. But when dealing with larger, more complex formulas, these extra steps can be the difference between being able to prove a goal within the maximum search depth, or not being able to do so. Therefore, the tactic will also try to prove the unabbreviated version as a second option. Because of the fact that the first option is more likely to succeed (at least it was on the case studies), the second option is tried last. As we can see in the code above, this second option is equivalent to the first with the unabbreviation and additional simplification left out.

One might wonder at this point why the second option is not used first, since it will run in shorter time if both are unsuccessful. The reason is that in the case studies the first option is much more likely to be successful and the difference in time is more or less insignificant (as it only involves rewriting tactics). So using the currently implemented order will give a slightly faster tactic in most situations. The rare proof in which only the second option succeeds is slowed down.

4.5 Satisfiability of implicit definitions

Another group of proof obligations is caused by the use of implicit functions. Since implicit functions do not have a description of how to calculate them, the question is raised whether there actually is a way to calculate them. To ensure there is one, a proof obligation is generated, stating that the postcondition of the function must be satisfiable. So for example, if we have the following implicit function definition:

```
sqrt (x : real) r : real
pre x >= 0
post r * r = x;
```

Then the following proof obligation would be generated:

$$\forall_{x:real}.\text{pre\_sqrt(x)} \rightarrow \exists_{r:real}.\text{post\_sqrt}(x, r)$$

Which can be rewritten to:

$$\forall_{x:real}.x \geq 0 \rightarrow \exists_{r:real}.r \cdot r = x$$

A satisfiability obligation is also generated for a 'let be such that' expression. Since this expression shows much similarity with the implicit functions definition (a 'let be such that' can directly be rewritten to an implicit function definition and vice versa), it belongs to the same category.

Due to its 'satisfiability nature' it looks a lot like the problem we had when trying to define types during translation. In that case we had to find a witness value that would prove that the type was not empty. In this case, we have to find a witness value for every parameter to prove that the function can be satisfied. Since this is as difficult as finding a way to calculate the value itself, it is extremely hard to prove automatically. In rare occasions when dealing with a strange postcondition (e.g. true, or equivalent to the precondition), it may be possible. Otherwise it is simply not feasible to do this in a fully automated way at the moment.

4.6 Termination

The last group of proof obligation arises from the use of recursive function definitions. The value resulting from the evaluation of a recursive function is only defined if the function will terminate at some point. Each recursive function will trigger a proof obligation to ensure this. Proof obligations belonging to this class are therefore relatively rare. There were no recursive functions in the case studies, so no termination obligations were encountered there. There have been some successful attempts to use recursive functions outside the case studies.

Any function of our model will be defined through the 'Define' function of HOL (see Section 3.5). Similar to the termination proof obligations, the 'Define' function requires functions to terminate. Whenever it is called with a recursive function as its parameter, it will try to prove its termination. If successful, the theorem of the definition of the function also states the termination of the function. Using this theorem, the termination proof obligations can quite easily be proved to be valid (in one or two steps). Unfortunately, if the 'Define' function is not able to prove termination, it will not define the function at all and request a termination proof from the user first.

There is currently no support built-in to handle the situation that the Define function cannot prove the termination. Since the proof has to be found automatically, such a scenario asks for a better tactic than being used in the Define function. Since an entire thesis can probably be devoted to this topic, no further attempts have been made on this topic. Furthermore, the current tactic in Define is rather strong. It can prove many recursive functions to terminate. Best practice when requiring a full automated proof of all obligations seems to be to prevent the definition of too complex recursive functions. When a function becomes too complex is, without extensively studying the tactic in 'Define', hard to say. A user guided proof of the termination of a function is of course always an option, yet not an easy one and certainly not one within the scope of this thesis.

Chapter 5

Case studies

To ease the development of the tactics and to assess the quality of the system, several case studies have been used during development. When looking at the performance of the tactic on the various case studies, it is important to keep in mind what the original intention of the case study was. Three types of case studies have been used:

- A series of simple case studies developed for this project. The main intention was to test the translation model with them. They were never meant to test the performance of the tactic and since they usually trigger no proof obligations, they are not useful to do so.
- A series of case studies from the PROSPER project. These cases are specifically meant to be hard to prove. They usually use all kinds of constructs and provide situations that are rare in practice but push the maximum out of the proof attempts. Although they are relatively useful to improve the tactic, they are not very useful in assessing its quality in terms of the number of proofs successfully completed.
- One case study has been used that was not specifically altered or prepared for the use for this project. This is the Mondex-abstract case. The number of different constructs is normal and it can be considered as a realistic example. It has been chosen for the fact that it still generates a lot of proof obligations.

Using a single case study is in general time-consuming. The main cause is the amount of manual work required to get a case study to a format suitable for proof attempts. The translation may be automated, starting the translation with the right input is not (yet). Furthermore, trying to find what went wrong in proofs and trying to improve the tactic to solve this, is extremely time-consuming (but as we will see, also very useful). In general, converting a case study to the right format takes from a few minutes to a few hours depending on the number of proof obligations. Adapting the tactic can take several days (or of course be done right away, if all obligations are proved without having to improve the tactic).

The first category of case studies above will not be discussed further here. They were useful during development, but are not directly relevant anymore

after they have served their purpose. The second category will cover most of this chapter as most case studies belong to this category. The last category is unfortunately rather limited in number of cases. Clearly more would have given more insight into the power of the tactic and the quality of the translation, but the timing of the project only allowed for one.

This chapter has a section devoted to each of the case studies from the second and third categories. Their context, meaning and code-style will be discussed, as well as the proof obligations they trigger and the performance of the tactic on these. One case (the Safer case) is not discussed as it has only been used partially and no statistics, or significant conclusions can be derived from this case. Furthermore, its contribution to the development of the tactic is minimal.

Before looking at the cases themselves, we first discuss how to use them in current setting.

5.1 Usage of case studies

Although this project should eventually fit into the Overture project, most components that should eventually make up the Overture tool set are still missing. This has its effects on the current usability of Overture tools, but also its effects on the development process of its tools. As mentioned earlier, the lack of type information in the AST, when actually executing the code, more components might have been useful. The software components required to run a case study through the prover are listed below (ignoring the type information that might have been useful).

1. Parser
2. Integrity checker
3. Translator
4. Proof control
5. Proof system (HOL engine and tactics)
6. Feedback functionality
7. Connections between the components above

The first component was readily available in the Overture project from start on. The second component unfortunately was not. To generate the obligations, the integrity checker from the VDMTools set had to be used, after which the generated proof obligations had to be converted manually to be used in the translator (since automating this would have taken too much time). The translator itself (component 3) was developed within this project and therefore available when needed. Component 4 includes handling proof failures and proof successes as well as possibly choosing tactics or handling errors from the HOL engine. This component could have been developed in this project, but since the concrete setting of the overture tools is not 100% clear yet, this has been postponed. The component might have been useful in testing, but not directly crucial as most its tasks can easily be done manually. Component 5 was partly available in advance (the HOL engine) and partly developed during the project (the tactics).

Components 6 and 7 comprise the glue that should stick everything together. Component 6 should interpret the output of HOL and return in to the user in a readable format. Component 7 should provide connections to transport the right information to the right component. Most of these connections were not available in advance and developing a connection was not always in the scope of this project or time was simply too limited to do so. All missing connections were solved by manually doing the work they should have done.

Clearly the manual replacements of the missing components take up a lot of time and cause tests of proof obligations to take much more time than they would have cost if everything had been automated. Nevertheless, testing the case studies was doable and a valuable contribution to the project.

The use of the case studies themselves was mainly to improve the performance (in number of successful proof attempts) of the tactic. Evaluation of performance was done partly by simply looking at the number of successes relative to the number of attempts, but this was hardly ever sufficient to be able to improve the tactic. Most evaluation was therefore done by carefully examining the failed proof attempts, spotting the missing links in finding the proof, adding this link manually to see if this was indeed the missing part and then trying to generalize the link to add it to the tactic.

Although this process seems to be time consuming, it had a relatively high success rate. he generalization was crucial. Most fixed links provided more than one extra successful proof attempts over the various cases.

A proof attempt is called unsuccessful, if it fails to find a proof entirely, or if it fails to do so within about thirty seconds (on a AMD Athlon XP 2500+, 512 MB notebook). Clearly, this is not a strict and rather arbitrary limit, but as the HOL manual mentions, if HOL does not find a proof within a very short time, it probably will not find a proof in a longer time either (which is reasonable considering the branching factor of the decision tactics). It clearly does not rule out that it might have found a proof anyway after thirty seconds.

Note that although a boundary is set at thirty seconds, it rarely takes all thirty seconds. If no proof can be found, the tactic nearly always already fails within 10 seconds, because the maximum search depth has been reached. Increasing the search depth is of course an option, but this has in the cases that have been used here never led to more results.

5.2 Alarm case

The alarm case study is a small case study that is based upon a larger system controlling a chemical plant. The original system was introduced in [18]. The code of the case study covers about one page and contains five fairly straightforward functions.

5.2.1 Case description

The case models a schedule of experts being assigned to specific periods in time. Each expert has one or more qualifications. The schedule is used in a (chemical) plant, that can trigger different types of alarms. Whenever there is an

alarm, the right expert has to be notified (the one that is on duty and has the right qualification). Invariants have been added to assure many criteria, among which: every expert has some qualification, every expert has a unique id and at any period in time, for every alarm, there is an expert on duty that has the qualification to handle this.

The case already contains a wide variety of language constructs, such as: maps, sets, records, invariants on types, implicit as well as explicit function definitions, set comprehensions and pre- and postconditions. It consists of seven types and five function definitions (of which two are used in the invariants).

5.2.2 Proof obligations

Surprisingly, despite the complexity of the case, it only results in two proof obligations. One is caused by the use of a map application and therefore belongs to the category of domain checking proof obligations. The other is a satisfiability of a postcondition of an implicit function definition in the model. The first proof obligation can be solved as a domain checking obligation right away (using only the most basic tactic discussed in the previous chapter). The second cannot be proved using current tactics at all. As explained, satisfiability is usually too complex to be proved automatically.

5.3 Memory case

The memory case was the second case study that was originally used in the PROSPER project. Covering about two pages, it is slightly larger than the Alarm case.

5.3.1 Case description

The memory case models exactly what its name indicates. It stores the data contained in the memory, information about what part of the memory is in use and it can lock the memory cells while they are being written to. The major part of the case holds a set of allocation functions that all allocate memory in a different way. The remainder of the model defines functions to verify the consistency of the memory.

In addition to the language constructs used in the alarm case study, the memory case also uses a map override, map union and a 'let be such that'.

5.3.2 Proof obligations

The proof obligations of the memory case are much more interesting than the mere two of the Alarm case. There are 20 in total. The division over the different proof obligation categories can be found in Table 5.1.

Using the discussed tactic that included rewriting of definitions in goals, eventually all proof obligations from the first and last category could be proved. But the second category exposed a more complex problem that could not be solved merely using the tactic. Take for example the following types and allocation function from the model:

Category	Number of proof obligations
Domain checking	6
Subtype checking	9
Satisfiability of implicit definitions	5

Table 5.1: Division of proof obligations in the Memory case

```
ADDR = <a0> | <a1> | <a2>;
CON  = <c0> | <c1> | <c2>;
Alloc : ADDR * State -> State
Alloc (addr, mk_State(mem,access,used)) ==
  let
    used' = used union {addr},
    mem'  = mem ++ {addr |-> let c:CON in c}
  in
    mk_State(mem',access,used')
pre addr not in set used;
```

The first two lines define the enumeration types that hold the addresses of the memory and the content that one memory cell can have. Named `ADDR` and `CON` respectively. The type State holds a memory state, containing the memory cells themselves, a set of locked cells and a set of cells that are in use. It has the following invariant:

```
inv mk_State(mem,access,used) ==
  used = dom mem;
```

The allocation function, given an address, updates the state in such a way that the address is in the used set and an arbitrary value is stored at it using the map update (++). A precondition is added to make sure the cell was empty and thus nothing is overridden.

Among others, this will generate the following proof obligation (using VDM patterns to keep the formula simple):

$$\forall_{addr:ADDR,\text{mk\_State}(mem,access,used):State} addr \notin used \quad \rightarrow \quad \text{inv\_State}(genState)$$

in which *genState* is the state that is generated in the function. Clearly, in the generated proof obligation this *genState* variable is not used and its entire definition is put into the formula. However, to keep it readable, this is all that is directly important here.

We have to prove that the used set in the generated state (called `used'` there) is equal to the domain of the memory in the generated state (called `mem'` there). Since the value addr of the function is added to the used set, as well as to the domain of the mem set, this property holds if we merely look at this value. Furthermore, the entire proof obligation holds if the state invariant is satisfied by the state parameter of the function. Which holds due to the signature of the function. Unfortunately we did not know this in HOL, since all invariant information was left out of the HOL type definitions. So, by removing type information from the type definition, we have not only made the types

less restrictive, we have also made proving some obligations impossible. To solve this problem, we can add the additional invariant information to the proof obligation, as we have explained in Section 3.5.2. Therefore, the proof obligation that is being proved in HOL is the HOL version of:

$$\forall_{addr:ADDR, \mathrm{mk\_State}(mem,access,used):State} \mathrm{inv\_State}(\mathrm{mk\_State}(mem, access, used)) \wedge addr \notin used \rightarrow \mathrm{inv\_State}(genState)$$

Using this additional type information, six more proof obligations of this case were proved automatically (these proofs are usually rather complex and may require a few seconds to be found). Three proof obligations remained. A good inspection of them surprisingly revealed that they could not be proved at all. They simply were not valid, since the model used as case was inconsistent. In many situations this would render the model useless for testing, but in this case, it showed that indeed errors in the model can be found.

All together, all obligations that are valid can be proved automatically.

5.4 Tracker case

The tracker case was also used in the PROSPER project. It models a tracking system of containers in a nuclear plant. It is also about two pages in length, but more complex than the previous case.

5.4.1 Case description

All containers that are being tracked in the model can pass through several phases. Of each container the quantity of the material it contains as well as the type of material is registered. Each phase has one or more types of material it can handle and a certain maximum quantity of it (its capacity). Containers can be moved in between the phases, but a container is not allowed to be in a phase that cannot handle the content of it.

There are functions available to introduce new containers to the plant and to remove them, to add containers to phases and to remove them and to move containers around. In addition to these, three functions (Consistent, PhasesDistinguished and MaterialSafe) state the properties that need to be satisfied to make the system safe. For this reason, these are used in the plant invariant (which is a record type).

Added language constructs in this case are several set operators (such as difference) and several map operators (such as map domain and range restrictions). All together, the case makes use of nearly all language constructs that can be translated.

5.4.2 Proof obligations

This case is a specific example of a rather complex system that may be rare in practice. The magnitude of the invariants and the number of restrictions within

the functions do not directly provide a high number of proof obligations, but more complex proof obligations. More complex in size as well as in the difficulty (length) of their proof. Their division can be found in Table 5.2.

Category	Number of proof obligations
Domain checking	7
Subtype checking	6
Satisfiability of implicit definitions	0

Table 5.2: Division of proof obligations in the Tracker case

Of these obligations, all domain checking proof obligations can be proved automatically, but the subtype checking obligations are more complex. One could be proved, but even when including the additional invariant information discussed in the previous section, a proof of the others cannot be found within the thirty seconds. When examining these proof obligations more closely, it is not hard to realize that finding a proof must be hard to be done automatically, as it is hard to do manually too. The hard parts of the obligations usually involve proving that the capacities of phases are not exceeded or that all containers are in a safe location (i.e. not in a phase that cannot handle the materials in the containers).

Performing this case study has especially improved the handling of maps through the addition of custom and built-in theorems. Not only the eventually proved obligations were useful, but also the ones that could not be proved. Since, they may not have been proved entirely, but many sub goals of them can and have been proved automatically after improvement of the tactic. Although it might not show any result on the high level statistics here, it hopefully will have effect on the proof of obligations from other cases.

5.5 Mondex case

The Mondex case was the last case to be tried. It models an electronic purse system and covers about two pages.

5.5.1 Case description

The Mondex case is slightly different from the other cases as it was not specifically chosen to be complex, or to generate many proof obligations. In that, one could say that it is a more realistic case, but also slightly less useful to develop the tactic. The entire Mondex case holds a concrete as well as an abstract model, but due too time limitations, only the abstract variant has been used.

The Mondex case itself models 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 the world and to verify transfers of money.

No additional language constructs are used by this case. In general, it uses much less types of constructs than the previous cases[1].

[1] This is probably directly related to the developer of the model, not to the case itself

5.5.2 Proof obligations

In general, the 30 proof obligations of this case are not extraordinary complex and not extraordinary simple. The obligations themselves show no extraordinary features or requirements. Their division can be found in Table 5.3.

Category	Number of proof obligations
Domain checking	23
Subtype checking	7
Satisfiability of implicit definitions	0

Table 5.3: Division of proof obligations in the Mondex case

All proof obligations can be proved automatically. Four of them require the previously discussed type information. One proof obligation could originally not be proved, since it was incorrectly generated by the integrity checker of VDMTools. This bug has been reported and the obligation that it should have been *can* be proved automatically.

Chapter 6

Related work

This chapter will focus on three related projects. The PROSPER project is most related and will be discussed extensively. A general description, a comparison with this thesis and a comparison of results will be given. When discussing the results, we will only look at the results that are relevant in the context of this project. The last two sections will look at the same type of work applied to different languages and using a different theorem prover.

6.1 PROSPER

The aim of the PROSPER project was to overcome the barrier between regular system development and the use of formal proofs to aid in this process. Existing proof tools require a lot of expertise to operate and do not integrate into the regular design flows. Yet with the increasing demand of high software quality, they may just be able to provide the missing link. To achieve this, two prototype tools were developed in PROSPER. One suitable for hardware modeling and one for software. Both used the HOL engine HOL98 for proof support. The hardware variant focused on the industry-standard languages Verilog and VHDL. The software variant used VDM-SL as base language. Both had the aim to provide user guided proof support as well as automated proof support.

Especially the software variant is of interest here. The automated proof part of PROSPER was similar to this project, but in practice had some differences in objectives: First of all, PROSPER focused on the VDM-SL language, whereas this project focuses on the extension of this, namely VDM++. Second of all, PROSPER used the older version of the HOL engine (HOL98). Third of all, PROSPER used the VDMTools toolbox as starting point, whereas this project uses the Overture toolbox as starting point.
These three differences are relatively small: The VDM-SL language is rather similar to the VDM++ language and has most language constructs in common, the upgrade from HOL98 to HOL4 did not change much of the base structure of HOL and the added libraries were small in number compared to the ones already available and in this project the integrity checker of VDMTools needed to be used anyway, as an Overture variant has not been implemented yet.

Unfortunately most PROSPER sources and documents were lost. What was left were some documents in their development stage, some case studies and

most of the HOL98 libraries written for PROSPER. Despite of these documents, most of the PROSPER ideas were hard to recover. Therefore not many ideas of the PROSPER project could be used and some of the choices made have been made differently in this project as we will see in the section below.

6.1.1 Similarities and differences

The documents left of the PROSPER project were on one hand limited in explanation, leaving much up to the imagination of the reader and on the other hand incomplete since they were development versions. A full comparison of the PROSPER project and this one is therefore not feasible. Yet, we can take a look at the ideas that were recovered from the documents and HOL libraries that were left over. This section will look at several topics in the project that can in some way be compared to PROSPER.

Before looking at the differences themselves, it is worth noting that most were not caused by the difference in used theorem prover versions (HOL98 versus HOL4). The theorem provers HOL98 and HOL4 were mostly similar. The only differences apply to certain libraries that were mainly left untouched in the PROSPER project.

Translation method In contrast to this project, the translation used in PROSPER was split up into two separate parts. At first there was the type of translation as has been used here: VDM-SL models to a HOL specification. But this was in general incomplete. Several of the VDM-SL constructs were translated literally, even though they did not have a literal translation in HOL. The second part of the translation consisted of a HOL library. This library completed he translation by providing a set of additional definitions.

The focus of this project has been mainly to keep the entire (split-up) translation done in PROSPER on one side, which has to be either the VDM or the HOL side. Since a translation cannot purely exist on the HOL side (since a VDM model cannot without translation be read into HOL), the VDM side was chosen. This way, when the need arises to change the translation, or add constructs, only one location has to be changed. Using this method no additional libraries are required anymore for the translation. All functionality is incorporated in the VDM++ model of the translation.

One might consider specifications on the HOL side to be more 'formally secure', in the sense that all theorems on that side have to be proved. But that is only partially true. Indeed, theorems have to be proved in HOL, but definitions can be inserted anyway. There is no need or reason (or even a possibility) to prove definitions. This is merely an introduction of a new concept (usually a function or a predicate). This means that definitions on the VDM side are basically just as insecure (and secure) as definitions on the HOL side.

Pattern management In contrast to most other parts of the project, the handling of patterns is relatively similar. The implementation (especially using two types of pattern evaluations) may be rather different, the principles are up to a certain extend similar. Both projects use rewriting of the VDM model to a new VDM model and both use the extraction of free variables (partly because this is probably the only solution). But the actual assignments using patterns is

never explained in the PROSPER documents and might have been implemented completely different. The comparison of patterns is never mentioned and therefore probably never used. There seem to have been a different approach to solve that.

Type management One of the major difficulties of this project was the handling of types. Especially invariants are hard to cope with. In PROSPER they most likely have decided to define two additional HOL meta level types next to the existing hol_type:

- A VDM type, which holds values directly resembling all VDM types ignoring any invariants
- A combination of an invariant (predicate) and the above VDM type

Although this distinction may at first sight seem valid and useful, actually using it is unnecessarily complex (at least, in the author's opinion). The problem is that these two types are not an integral part of the HOL engine or the HOL libraries. To use them, much additional code has to be written. Not only to construct values of the types, but also to extract information from them or even to use them. This additional writing is not only a lot of work, but also a cause of errors. Nevertheless, the PROSPER project seems to have managed to complete most of it.

In this project, the type management has been focused on using as much of the HOL functionality as possible. Only HOL types are used and invariants are translated explicitly into the model, without any need for additional HOL libraries to deal with them. This reduces the amount of work, reduces the risk of making errors and most importantly increases the usability of HOL libraries dramatically. In PROSPER, most of the theorems that are primitively available in HOL libraries, had to be redefined for the newly introduced VDM types. Thus reducing the number of available theorems (not all theorems were used) and removing the possibility to make use of new versions of libraries.

PROSPER tactics The original intention of PROSPER was to translate the available HOL98 libraries of PROSPER to HOL4 libraries and use them again, but mainly as a result of the differences in translation, the libraries have lost their applicability in the current project. They were redesigned and reimplemented. However, the differences between the resulting tactics of the two projects are enormous.

Since the translation in PROSPER made limited use of the HOL possibilities, the tactics had to follow this recipe. As we have seen, there are various useful tactics available in HOL. Most of these were used, but only in a rather limited ways. Clear similarities can be spotted between the domain checking tactics of both projects. But from the subtype checking tactics on, there is hardly any resemblance left. Where this project uses HOL tactics that are customized by the use of additional theorems, the PROSPER project uses a completely custom-made tactic. This tactic backtracks over the proof search space, trying to decompose and prove the goal along the way. Since most HOL tactics use a similar approach and are likely to be more powerful (as most likely more time has been put into their development), it is hard to say what the arguments were

to choose this approach. Yet this judgment is based on the limited information available and there might be reasons that simply have been lost.

Another striking difference is the lack of rewriting that has been used in PROSPER. Even though the HOL rewriting systems are fast and relatively powerful, only limited use of the rewriting can be found. When compared to the usefulness of the large amount of rewriting used in the tactics of this project, it is hard to say what the arguments for this lack of rewriting attempts were.

6.1.2 Results of PROSPER

Having discussed the differences, it would be useful to compare the performances of the two approaches. But although the project goals were more or less the same and although the same case studies have been used, a comparison is rather complex. The main reason is that there seems to be a difference in the proof obligations that were obtained on seemingly the same case study. At least they differ in number (which can be found in the PROSPER documents) but they might then even differ in semantics. Since this is the case for all case studies, there are at least two reasons that would explain the difference:

- The integrity checker used in this project discards some obvious proof obligations, while it did not in the time of PROSPER (a few years ago)
- The case studies extracted from what was left of the PROSPER documents were not the ones used during evaluation of the PROSPER implementation

Either way, a comparison with decent results cannot easily be made anymore. If we compare them anyway, knowing that there are differences in the process or the cases, the conclusion is that the performances (in fraction of successful proofs) is more or less similar. But this is merely an estimate and no further conclusions should be drawn from it. It is possible to say that both approaches yielded good performances and looked promising.

On the topic of computational performance (execution time), we cannot make a comparison either. Although the PROSPER documents mention exact execution times for many of the proof attempts, they do not state what computer was used to execute them.

There is one case that has only been mentioned briefly in the introduction. This is the Interlocking case, in which the PROSPER project was able to prove about 90% of the proof obligations automatically. This may be a good candidate for comparison (although a clear description of the results is not available), but there is not enough time left in this project to try the case.

6.2 TIAPS

Preceding the PROSPER project, the TIAPS project (Towards Industrially Applicable Proof Support for VDM-SL) [11, 12] attempted to achieve a similar goal as in PROSPER, using the general-purpose theorem prover Isabelle [9]. TIAPS was a joint project of IFAD and the Technical University of Denmark (DTU), which only intended to develop a prototype. In contrast to the PROSPER project, the translation and proof support in TIAPS were not built around the generated proof obligations, but instead, the generation of the proof obligations

was developed at the same time and designed to provide obligations suitable for TIAPS. The Isabelle proof tool is in many ways similar to HOL, reducing the differences between the TIAPS project, PROSPER project and this project. In contrast to HOL, Isabelle does have a graphical user interface, that was used throughout TIAPS to ease interactive proof.

6.3 Java program verification

A more recent project [34] focuses on Java program verification. In order to get the formal specification basis in Java required for proof, the extension Java Modeling Language (JML) is used, which is a behavior specification language for Java programs. The project focuses primarily on iterative programs and only the properties that were specified in JML are verified. To get the proof obligations from a JML annotated Java program, the program is translated to the input language to the Why program [7] by a tool called Krakatoa [10]. The Why tool will then generate proof obligations from the translated Java program. Just like in this project, the theorem prover HOL4 is used in an attempt to proof the obligations. The output from the Why tool can directly be read by HOL.

In contrast to this thesis, no automated proof has been developed. So far, the project has focused on manual verification, still requiring a lot of effort and knowledge from the user. Only a small example is being used. Once larger examples are given and the number of proof obligations increase to much higher numbers than the five that were proved in a given example, users will probably not be very tempted anymore to prove all manually. Yet, the project does provide a means of starting automated verification of Java programs using HOL.

6.4 C# program verification

A more developed project, yet somewhat more distant from the methods used here is Spec# [5]. Spec# is a programming system which provides an extension of the regular C# language, introducing pre- and postconditions and invariants. Furthermore, it provides automated verification of these properties using a theorem prover. Currently the Simplify theorem prover is used, but the aim is to switch to an experimental theorem prover developed at Microsoft research [14]. The proof obligation extraction as well as the verification follows the object oriented approach laid out by the C# language. Furthermore, the results of the proof are being mapped back onto the source code for user evaluation.

Chapter 7

Conclusions & Further research

7.1 VDM to HOL Translation

About half of the project consisted of writing a translation of VDM to HOL. When the project started, the intention was to build the translation in several iterations, looking at a new case study every iteration and eventually ending up with a translator that was just able to translate the most difficult case. However, in practice it soon became clear that to translate the smallest case study that would give proof obligations, the translation needed to be rather complete. The smallest case already required about 75% of the code of the final version of the translator. Approximately another 25% was required to support the other case studies. Because of the fact that each case study requested this much code from the translator, the translation is rather complete. At least, when only considering the non-object-oriented, functional subset. The language constructs that are not supported yet (mainly operators and pattern types) can be added without altering the current framework.

With respect to the computational performance of the translation, it is fair to say that the system is relatively slow. The main reason for this is the dependency resolution, that first of all uses a slow sorting algorithm (selection sort) and second of all has a lot of overhead when determining dependencies. Increases in performance are straightforward and relatively easy to implement.

The quality of the translation itself is an entirely different topic. Specifically the semantical equivalence between the two models is crucial to the proof. There are two aspects that are of importance:

- The correctness of the principles of the translation
- The correctness of the implementation of the translation

The first has been the topic of a large part of this document. It can be judged upon the text that is written here. Yet, the second aspect is much more complex. The number of errors made in the implementation is hard to determine and the implementation usually has to deal with many details that were left untouched in this document. Considering the case studies of this project, it is

possible to be relatively sure that the translation is correct on them, as these have been checked manually. But on an arbitrary case study, the quality is hard to predict based on a small set of tests (10). A larger base of cases could be used to do additional testing, but the result would have to be verified manually. A time-consuming and tedious job, but still an effective solution.

All together, except for some small unimplemented parts, the goals stated have been reached and the challenges have nearly all been faced and solved. Only the object orientation could not be solved within the available time. The result is a well functioning model, that might require some relatively simple improvements in computational efficiency and some additional testing.

7.2 Proof of obligations

The proof attempts, which covered the remaining time of the project are in many aspects very different from the translation. Instead of a mapping from a known source to a restricted target, the proof attempts are a search for a sequence of which only very limited knowledge is available. This changes the approach that has to be taken to solve the problem and introduces a trade off between time and effectivity.

The goal of this part of the project was to prove as many proof obligations as possible. Since there are still proof obligations in some of the case studies that might be proved using an improved tactic, the goal has not entirely been reached. Let us therefore take a look at the overall results of the tactic on the case studies used. Table 7.1 shows the results summed over all proof obligations. As we can

Category	# valid obligations	# proved
Domain checking	37	37
Subtype checking	19	14
Satisfiability of implicit definitions	6	5

Table 7.1: Summed scores of each of the case studies used

see, the domain checking proof obligations can all be proved. The subtype checking obligations show a 74% score for these cases and of the satisfiability all except one could be proved too, though, it is useful to keep in mind that most of these were relatively trivial.

Judging on the scores we could directly say that the automated proof is very useful to the user. Most of the proofs are generated automatically. Those that are not, can be proved manually after all (or the corresponding part of the model could possibly even be rewritten to prevent the obligation). It is not directly clear from the numbers that the tactic was adapted to fit these cases. The current tactic is therefore not likely to reach such high scores on completely new case studies (although it still reached a 70% when looking at all of the proof obligations of the Mondex case when it was completely new). Based on the current results, we are convinced that when the tactic is improved using several new case studies, the score will be this high on any new, 'more or less regular' case study, but tests would have to verify this.

Two of the three challenges in this part of the project were focused on execution time of the tactic, as this seemed to be one of the major problems in

advance. Therefore the tactics were developed in a way that they remained time efficient. Because of this, the actual execution time is short and there is no need to improve the tactic much at this point. Clearly there were a lot of concessions during development of the tactic (only a very limited number of theorems is used), but this clearly paid off, more than originally expected. The main advantage is that this allows for much extension of the tactic without losing its usability.

Judging on all knowledge gained throughout the thesis, the automated proof is without a doubt successful on the case studies used. Not only judging on the scores, but also judging on the execution time. The tactic's applicability on completely new case studies, is hard to determine without trying many more case studies, but is most likely sufficient. It can however be made much more effective easily when improving the tactic with additional case studies.

7.3 Further research

Several of the issues that may require further research or further implementation have already been mentioned before. Below I will give a list of all issues that were left open, or are useful (or crucial) extensions of what is currently available:

Code completion The current implementation works fine, but many tasks still have to be performed manually and it undoubtedly still contains bugs. Especially the connections between the various components (such as parser, translator, HOL and user interface) require additional work and exhaustive testing may be useful. The current implementation may function, but it certainly is not completely safe and usable yet.

In addition to filling in the missing parts, also completion and improvement of efficiency of the translation is required to make it usable.

Extension of translation If the current code functions correctly, extensions may be useful to provide additional model support. The main extensions are the object orientation constructs. Most likely they can be added without changing much in the proof support and without having to change much in the translation. Additional type information in the AST might be useful as a prerequisite.

Extension of tactic Even without extension of the current translation, extension of the tactic may be useful. Due to the used case studies, the current focus is mainly on maps and sets, but there are clearly more constructs in the language that can generate proof obligations. Of the three categories, the first (domain checking) will most likely always or nearly always be proved automatically using the current tactic. The third category (satisfiability) will most likely be proved too, or simply be too difficult to prove. That leaves especially the second category (subtype checking) as a very interesting category. Many proofs can probably be found in this category by simply extending the current tactic with additional theorems. There is no need to change its structure yet, an extended base of rewrites and an extended base of inference rules should increase the number of successful proofs.

New concepts When translation and tactics function correctly, new concepts can be introduced. New concepts that are at this point of direct interest are:

- *Allowing operational language components* This will not only affect the translation, but especially the proof which will be more complex. Since HOL has support for operational semantics, it should be doable. Otherwise, one of the operational approaches mentioned in the Related Work chapter could be used.
- *Allowing validation conjectures* Current implementation will only focus on proof obligation that were a direct result of the used constructs (such as type invariants and pre- and postconditions). In addition to these it might be possible to state validation conjectures over the model. These are predicates that always hold during execution of the model (similar to invariants). Proving these more general expressions has not been tested. Depending on the conjecture, the current tactics will be more or less suitable to find a proof. Additional research may extend the tactics to make them more widely applicable to include support for this type of obligations.
- *User VDM feedback* One of the topics the PROSPER project looked at, that was not part of this thesis, is feedback to the user through the VDM model. A translation of the HOL specification back to the VDM model may indicate exactly what the proof is about in VDM terms which are understandable by the user. This involves more than just a translation, since it might be nice to also indicate exactly where in the model various parts of formulas originate.
- *User guided proof* As the automated proof will probably not be able to prove all provable obligations in the near future, user guidance in the proof may save the user a lot of work. Giving hints is a lot easier than giving entire proofs. This holds especially for witness proofs, such as proving satisfiability.

Bibliography

[1] HOL Kananaskis-4. `hol.sourceforge.net`.

[2] Moscow Meta Language. `www.dina.kvl.dk/ sestoft/mosml.html`.

[3] ProofPower home page. `www.lemma-one.com/ProofPower/index`.

[4] PROSPER Website. `www.dcs.gla.ac.uk/prosper`.

[5] SpecSharp. `research.microsoft.com/projects/specsharp`.

[6] VDMTools web site. `www.vdmtools.jp/en`.

[7] Why is a software verification tool. `why.lri.fr`.

[8] Overture - Open-source Tools for Formal Modelling. `www.overturetool.org`, 2007.

[9] Isabelle. `isabelle.in.tum.de`, 2007.

[10] The Krakatoa Tool for Java Program Verification. `krakatoa.lri.fr`, 2007.

[11] S. Agerholm. Translating specifications in vdm-sl to pvs. In *TPHOLs '96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 1–16. Springer-Verlag, 1996.

[12] S. Agerholm and J. Frost. An isabelle-based theorem prover for vdm-sl. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 1–16. Springer-Verlag, 1997.

[13] S. Agerholm and K. Sunesen. Reasoning about VDM-SL proof obligations in HOL. Technical report, IFAD, 1999.

[14] M. Barnett, K.R.M. Leino, and Wolfram Schultei. The SpecSharp programming system: An overview. In *CASSIS*, 2004.

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

[16] J. Bowen. Formal Methods. `http://vl.fmnet.info`, 2007.

[17] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[18] J. Fitzgerald and P.G. Larsen. *Modelling Systems - Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.

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

[20] J. S. Fitzgerald. *The Typed Logic of Partial Functions and the Vienna Development Method*, pages 427–461. EATCS Texts in Theoretical Computer Science. Springer, 2007.

[21] M. Gordon, R. Milner, and C.P. Wadsworth. A Mechanised Logic of Computation. *Lecture Notes in Computer Science*, 78, 1979.

[22] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, pages 11–20, 1990.

[23] *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*. International Organization for Standardization, 1996. International Standard ISO/IEC 13817-1.

[24] Cliff B. Jones. Scientific Decisions which Characterize VDM. In *FM'99 - Formal Methods*.

[25] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[26] R. Milner, M. Tofte, and R. Harper. The Definition of Standard ML. *The MIT Press*, 1990.

[27] M. Norrish, K. Slind, et al. The HOL System Description - For HOL Kananaskis-4. `hol.sourceforge.net/documentation.html`, 2007.

[28] M. Norrish, K. Slind, et al. The HOL System reference. `hol.sourceforge.net/documentation.html`, 2007.

[29] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[30] R.S. Pressman. *Software Engineering - A Practitioner's Approach*. McGrawHill Publishing Company, 2000.

[31] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[32] M.E. Stickel. A PROLOG Technology Theorem Prover. In *IEEE 1984 International Symposium on Logic Programming*, 1984.

[33] F.W. Vaandrager. Does it pay off? model-based verification and validation of embedded systems! In F.A. Karelse, editor, *PROGRESS White papers 2006*. STW, the Netherlands, 2006. ISBN-10: 90-73461-00-6, ISBN-13: 978-90-73461-00-0.

[34] A. Wang, H. Fei, M. Gu, and X. Song. Verifying Java Programs By Theorem Prover HOL. *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, pages 139–142, 2006.

Appendix A

The HOL subset

Below a syntax description of the range of the translation is given. This syntax does not only describe HOL terms. It describes the entire range of output. That includes ML expressions and hol_types.

Hol_{sub}	=	complexTypeDef | functionDef
complexTypeDef	=	'hol_datatype(', typeIdentifier, '=', complexType, ');'
functionDef	=	'Define(', identifier, (identifier [':', simpleType])*, '=', expression,');'
expression	=	identifier | application | conditional | lambdaAbstr | let | quantifier | recordField | recordUpdate | select | setComp | setEnum | tuple
simpleType	=	functionType | listType | mapType | setType| unionType | nativeType | typeIdentifier
complexType	=	recordType | quoteType
application	=	expression+
conditional	=	'if', expression, 'then', expression, 'else', expression
lambdaAbstr	=	'\', (identifier [':',simpleType]), '.', expression
let	=	'let',identifier,'=',expression,('and',identifier,'=', expression)*,'in',expression
quantifier	=	('!'|'?')',(identifier,[':',simpleType])+,'.',expression
recordField	=	expression, '.', chars
recordUpdate	=	[expression,'with'],'<|',(chars,':=',expression,';')+,'|>'
select	=	'@', identifier, '.', expression
setComp	=	'{', expression, '|', expression, '}'
setEnum	=	'{',(expression,';')*,'}'
tuple	=	'(',expression, (expression,',')+,')'
identifier	=	chars
functionType	=	simpleType, '->', simpleType

listType	=	simpleType, 'list'
mapType	=	simpleType, 'map'
productType	=	simpleType, ('*', simpleType)+
setType	=	simpleType,'set'
unionType	=	simpleType, ('+', simpleType)+
nativeType	=	bool | ind | num | real | char
typeIdentifier	=	chars

In which any occurrence of 'chars' means a sequence of (non-reserved) characters. Furthermore, most of the brackets are left out, to keep the syntax readable. Brackets will always be added to prevent ambiguity. This will always be around an expression or a simpleType.

The set hol_type, which is referred to throughout the thesis is a type defined by HOL. The subset of hol_type that will be used in this thesis has been defined above. It is called 'simpleType' there. The same holds for the HOL terms and ML statements that are being referred to. The subsets of those are equal to 'expression' and 'Hol_{sub}' respectively.

Appendix B

VDM operator translations

These are the translation of all VDM operators that are currently being translated. The AST numbers that are being referred to are the numbers that are assigned to them in the AST generated by the Overture parser. Not all operators have been translated. These were required for the case studies and probably for most VDM models. Missing operators can be added easily if a translation is available and known.

B.1 Unary operators

All unary operators in the following table (VDM and HOL) are prefix operators.

AST #	VDM Operator	HOL equivalent	Description
0	card	CARD	Cardinality
11	dom	FDOM	Domain
14	rng	FRANGE	Range
16	~	~	Negation

B.2 Binary operators

All binary VDM operators are infix operators. The HOL operator equivalents can be prefix as well as infix. To solve this, all HOL infix operators have a $ sign advancing it, which will make them prefix in HOL. When using the implemented code, these prefix indicators will no longer be visible, as all prefix operators that can be written as an infix operator will be written as an infix operator to improve readability.

AST #	VDM Operator	HOL equivalent prefix	Description
0	union	$UNION	Set union
1	-	$-	Arithmetic minus
3	psubset	$PSUBSET	Proper subset
4	=	$=	Equality
5	not in set	(\x y . ~ (x IN y))	Not element of set
7	<	$<	Arithmetic less than
8	>	$>	Arithmetic greater than
9	<>	(\x y . ~ (x = y))	Unequal
12	\	$DIFF	Set difference
13	in set	$IN	Element of set
14	or	\/	Logical or
15	>=	$>=	Arithmetic greater or equal than
16	<-:	(\x y . DRESTRICT y x)	Map domain restrict by
17	and	/\	Logical and
20	inter	$INTER	Set intersection
21	:->	(\x y . RRESTRICT y x)	Map range restrict by
22	<=	$<=	Arithmetic less or equal
24	++	(\x y . FUNION y x)	Map override
26	subset	$subset	Subset
27	munion	FUNION	Map union
28	+	$+	Arithmetic plus
29	=>	$==>	Logical implication

Appendix C

Additional Theorems & Proofs

This appendix holds the additional theorems and their proofs that are being used in the tactic discussed in the next appendix. Each of the additional theorems is written down in HOL code and can be inserted directly into the HOL engine in this format. In general, each theorem definition has the following format:

```
save_thm
(
  "THEOREM_NAME",
  theorem
);
BasicProvers.export_rewrites(["THEOREM_NAME"]);
```

There are two outermost expressions: `save_thm` and `export_rewrites`. The first will store the theorem in the current session, the second will export it to the `srw_ss` simplification set. Both have the name of the theorem (which can be any arbitrary string) as a parameter. The second parameter to `save_thm` is the theorem itself. To construct this theorem (or to construct any theorem at all) we need to prove it. Therefore a call is made to the function `prove`, which will take a term and a tactic as parameters. Using the term, it will first create a goal (no hypotheses and the term as conclusion), after which it will apply the tactic to the goal. If successful, it will result in the theorem, if not it will simply fail. The call to `prove` looks like:

```
prove
(
  “term to convert into a theorem“,
  Tactic to prove the theorem
)
```

In addition to the theorem definitions, there is also one tactic definition in the beginning of the code, which is sometimes used to prove the theorems. This tactic definition requires the definition of some additional tactics (`VDM_REWRITE_TAC`, `VDM_FULL_SIMP_TAC` and `VDM_GENERIC_PROVE_TAC`). These have not been included here, they are defined in the next appendix. When executing the code these tactics need to have been defined before executing this piece.

```
<< Additional code to resolve the dependencies >>
<<     of the tactic below should go here      >>

val VDM_GENERIC_TAC =
(
  (VDM_REWRITE_TAC()) THEN
  (VDM_FULL_SIMP_TAC()) THEN
  reduceLib.REDUCE_TAC THEN
  VDM_GENERIC_PROVE_TAC
);
save_thm
(
  "UNION_SUBSET_FDOM",
  prove
  (
    ‘‘! a b. FDOM a SUBSET FDOM a UNION FDOM b /\
              FDOM b SUBSET FDOM a UNION FDOM b    ‘‘,
    VDM_GENERIC_TAC
  )
);
BasicProvers.export_rewrites(["UNION_SUBSET_FDOM"]);
save_thm
(
  "SUBSET_FDOM_FUNION",
  prove
  (
    ‘‘!a b c.a SUBSET FDOM b ==> a SUBSET FDOM (FUNION b c)‘‘,
    VDM_GENERIC_TAC THEN
    (REWRITE_TAC [FDOM_FUNION]) THEN
    (‘FDOM b SUBSET FDOM b UNION FDOM c‘ by
      (MESON_TAC [theorem "UNION_SUBSET_FDOM"]))
    THEN (PROVE_TAC [pred_setTheory.SUBSET_TRANS])
  )
);
BasicProvers.export_rewrites(["SUBSET_FDOM_FUNION"]);
save_thm
(
  "IN_FDOM_FUNION",
  prove
  (
    ‘‘! x A B. x IN FDOM A ==> x IN FDOM (FUNION A B)‘‘,
    VDM_GENERIC_TAC THEN
    (REWRITE_TAC [FDOM_FUNION]) THEN
    (PROVE_TAC [pred_setTheory.IN_UNION])
  )
);
BasicProvers.export_rewrites(["IN_FDOM_FUNION"]);
save_thm
(
  "IN_SUBSET",
```

```
  prove
  (
    ‘‘!x A. (x IN A) ==> ({x} SUBSET A)‘‘,
    VDM_GENERIC_TAC
  )
);
BasicProvers.export_rewrites(["IN_SUBSET"]);
save_thm
(
  "FAPPLY_IN_FRANGE",
  prove
  (
    ‘‘! m x. x IN FDOM m ==> (FAPPLY m x) IN FRANGE m‘‘,
    VDM_SIMPLIFICATION_TAC THEN
    (RW_TAC std_ss [finite_mapTheory.FRANGE_DEF])
    THEN (SRW_FULL_SIMP_TAC [])
    THEN VDM_GENERIC_PROVE_TAC
  )
);
BasicProvers.export_rewrites(["FAPPLY_IN_FRANGE"]);
```

Appendix D

HOL tactics

This appendix holds most tactics used throughout the thesis to prove obligations. Similar to the previous appendix, they are written in HOL code and can be inserted into the HOL engine directly. The code also includes definition of several lists of theorems that are being used in the tactics.

```
val tacticCustomTheorems =
[
  theorem "UNION_SUBSET_FDOM",
  theorem "SUBSET_FDOM_FUNION",
  theorem "IN_FDOM_FUNION",
  theorem "IN_SUBSET",
  theorem "FAPPLY_IN_FRANGE"
];
val tacticBuiltInTheorems =
[
  pred_setTheory.UNION_SUBSET,
  pred_setTheory.SUBSET_DEF,
  pred_setTheory.SUBSET_TRANS,
  pred_setTheory.UNION_SUBSET,
  finite_mapTheory.FDOM_FUNION,
  finite_mapTheory.FAPPLY_FUPDATE_THM,
  finite_mapTheory.FDOM_FUPDATE,
  finite_mapTheory.FUPDATE_COMMUTES,
  finite_mapTheory.FAPPLY_FUPDATE,
  pred_setTheory.IN_INSERT,
  pred_setTheory.SUBSET_INSERT,
  pred_setTheory.DIFF_SUBSET,
  pred_setTheory.IN_COMPL,
  pred_setTheory.IN_DIFF
];
val tacticSimplificationTheorems =
[
  FUNION_FUPDATE_1,
  FUNION_FUPDATE_2,
  FUNION_FEMPTY_1,
```

```
  FUNION_FEMPTY_2,
  FDOM_FEMPTY,
  finite_mapTheory.FUPDATE_COMMUTES,
  finite_mapTheory.FAPPLY_FUPDATE,
  finite_mapTheory.FAPPLY_FUPDATE_THM,
  finite_mapTheory.DRESTRICT_DEF,
  pred_setTheory.IN_COMPL
];

fun SRW_RW_TAC (thmList) = fn g =>
  let  val srwss = srw_ss()
  in   RW_TAC srwss thmList
g end;
fun SRW_FULL_SIMP_TAC (thmList) = fn g =>
  let  val srwss = srw_ss()
  in   FULL_SIMP_TAC srwss thmList
g end;

val VDM_ARITH_TAC =
(
  (numLib.ARITH_TAC) ORELSE
  (intLib.ARITH_TAC) ORELSE
  (realLib.REAL_ARITH_TAC)
);

val VDM_GENERIC_PROVE_TAC =
(
  tautLib.TAUT_TAC ORELSE
  (mesonLib.GEN_MESON_TAC 0 10 1 []) ORELSE
  DECIDE_TAC ORELSE
  VDM_ARITH_TAC
);

val VDM_SIMPLIFICATION_TAC =
(
  (SRW_RW_TAC []) THEN
  (FULL_SIMP_TAC std_ss [boolTheory.LET_THM]) THEN
  reduceLib.REDUCE_TAC THEN
  (RW_TAC (std_ss) tacticSimplificationTheorems) THEN
  (SRW_FULL_SIMP_TAC tacticSimplificationTheorems) THEN
  (SRW_RW_TAC tacticSimplificationTheorems)
);

val VDM_ADDITIONAL_TAC =
(
  VDM_SIMPLIFICATION_TAC THEN
  (
    (
      Q.UNABBREV_ALL_TAC THEN
      VDM_SIMPLIFICATION_TAC THEN
```

```
      VDM_GENERIC_PROVE_TAC THEN
      (PROVE_TAC (tacticBuiltInTheorems @ tacticCustomTheorems))
    )
    ORELSE
    (
      (PROVE_TAC (tacticBuiltInTheorems @ tacticCustomTheorems))
    )
  )
);
```