Engineering College of Aarhus
Computer Technology & Embedded Systems

Master of Science Thesis

An extended proof obligations generator for VDM++/OML

by

Augusto Silva

Supervisor: Peter G. Larsen

Aarhus, 2008

Resumo

A geração de obrigações de prova para verificar a consistência de especificações uma técnica muito usada em VDM. Estas obrigações de prova são geradas pelo verificador de integridade da ferramenta VDMTools. Contudo a terminação de funções recursivas não era contemplada neste verificador de integridade. Na primeira parte desta tese, o verificador de integridade foi extendido de maneira a produzir obrigaes de prova relacionadas com as funções recursivas, aumentando o espectro do gerador de obrigações de prova. Esta funcionalidade foi implementada no VDMTools e está agora a ser usada por especificadores em todo o mundo. A segunda parte da tese incide no uso de um provador de teoremas (HOL) para prover as obrigaes de prova para a terminação de funções recursivas. São apresentadas estratégias e receitas de modo a facilitar a tarefa de possíveis utilizadores desta nova funcionalidade do VDMTools.

Abstract

The generation of proof obligations to check for a specifications consistency is a technique that has been used in VDM for a long time. The proof obligations are generated by the integrity checker of VDMTools. However termination of recursive function is not contemplated in this integrity checker. In the first part of this thesis, the integrity checker was extended to produce proof obligations related with recursive functions and their termination, augmenting the already existing scope of the proof obligation generator. This functionality is already implemented in VDMTools and being used all over the world. The second part concentrates on the use of a theorem prover (HOL) to prove this generated proof obligations for recursive functions. Strategies and recipes are provided to facilitate the task of possible users of this new VDMTools feature.

Acknowledgements

First of all I want to thank my supervisor Peter Gorm Larsen for the all the insight, motivation and guidance that enabled the proper realization of this scientific research. In addition I have to thank Konrad Slind for the valuable input on HOL.

I would also like to thank my teachers José Nuno Oliveira and Luís Barbosa at University of Minho that opened the doors for this ”adventure” in Denmark.

I would like to thank my parents for believing that I should pursue a higher education and supported morally and financially my stay in Denmark.

And finally, I would like to thank all the people that were in contact with me during my stay in Denmark that in a certain way also provided inspiration for this work.

Augusto Silva

Contents

Chapter 1

Introduction

A introduction to the domains covered in this thesis is made on this first chapter. First starting by explaining what are formal methods in general and VDM in particular. An overview of program consistency and proof obligations is then made. Next a summary on recursive functions and their terminations is presented. Finally it is described how the thesis is structured.

1.1 Formal Methods

The term ”Formal Methods” applies to the use of a mathematical notation in the specification and the use of such specification phase as a basis for verified design of computer systems [30]. The employment of formal methods is motivated by the need to build correct systems. So formal methods are normally used in the development of critical software. Critical in a sense that human, environmental or economical hazards may result if the software fails, e.g. aircrafts, nuclear power plants and stock trade. The purpose of these methods is to provide techniques for precise formalization of computer systems and their validation and verification.

1.2 Vienna Development Method

The Vienna Development Method (VDM) is a formal method that consists of a collection of techniques for modeling, specification and design of computer-based systems. It is a model-oriented method and it is suitable to apply in a wide range of areas. The idea of making models to assist in the analysis of complex systems has been used in more ”traditional” engineering disciplines with great success. VDM is one of the oldest formal methods and one of the

most widely used. It has been applied to many industrial problems successfully [41, 33].

One key concept in VDM is abstraction [31]. A specification is a formal description of a model of the system and its behavior at an abstract level. Models tend to be much shorter then their implementation as they focus more on the intended behavior of the system as opposed to its implementation issues. So in specifications, abstract data types[1] are used instead of the ones on the target machine or programming language. In addiction, important properties of the model can be recorded as data type invariants, pre and post conditions to functions. Unlike the implementation, the ”simplicity” of the model provides a richer environment for reasoning about the system. Requirement or design flaws can be detected earlier in the developing process [33].

The normal approach to formal development using VDM is to specify the model in a specification language. The design phase and code are written in a normal implementation language. A stepped transition, called refinement, establishes a correspondence between the model and the implementation. The correctness of each step is then established by discharging *proof obligations*[2] associated with it.

VDM can be applied at different levels of rigor [7, 4, 18]. On one hand one can use mathematical proofs to validate key properties or even all functionality in a model. This level is the most rigorous one but also the most time consuming. Alternatively, a more lightweight version can be used. With this approach a model is constructed only to reason informally about its requirements or to be validated using tools. Gaining insight in the intended functionality of a product that is planned for future development is the most common use of this technique. There are also several ways of increasing confidence in a model without complete proof of the system.

Several specification languages have supported the VDM principles each one enables modeling of different types of systems:

- VDM-SL – a purely functional specification language that allows the modeling of sequential systems.VDM-SL has been standardized by the *International Organization for Standardization* (ISO) [29].
- VDM++ – an extension of VDM-SL that supports modeling of object oriented and concurrent systems. This thesis will focus on this language although only in its functional subset [17].
- VICE (VDM In Constrained Environments) – a further extension to VDM++ that supports the modeling of distributed and real-time systems [51].

[1] A brief introduction to VDM data types is present on section 2.2.

[2] These proof obligations are related with refinement and have nothing to do with the ones present on this thesis, for more information consult [30]

A ASCii and mathematics notation exists for VDM but the ASCii notation will be used throughout this thesis.

1.2.1 Tool Support

VDMTools

VDMTools is a closed-source set of tools that supports the analysis of models expressed in VDM [18]. It supports all three specification languages mentioned above. This tool includes a parser, a type checker, an interpreter with debugging functionalities. These aid in the construction and visualization of VDM models. In addition, there is a Java and C++ code generator that when applied to a type correct model yield a fast implementation of the VDM model. Furthermore all the functionality in VDMTools can be accessed by a CORBA API permitting a rapid prototyping of models with a GUI designed in C++ or Java. Additionally it is possible to generate of UML class diagrams from VDM models and vice-versa. It is also possible to make systematic testing of models and retrieve valuable statistics about what areas of the code have been tested.

The most important characteristic of VDMTools for this thesis it is its integrity checker. It generates checks to avoid semantic inconsistencies and possible run-time errors. These checks are called *proof obligations* and a deeper overview will be given in section 1.3. The goal of this thesis was to extend the previously existing VDMTools with another type of checks for termination of recursive functions.

Overture

The Overture Initiative is a community-based project committed to develop Open Source tools to aid in formal modeling and analysis in design of computer based systems. It relies on long time research and industrial application of VDM [39].

In modeling practice some features of VDM++ turned out to be rarely used, over-complex or difficult to understand. So the Overture Initiative and the Overture Modeling Language (OML) were born with the hope that a more usable modeling method could be created [50]. OML is virtually equal to VDM++.

The Overture mission is to to provide industrial-strength tools that supports the use of precise abstract models (expressed in OML) in software development. It is also meant to foster an environment that allows researchers and other interested parties to experiment with modifications and extensions to the tool.

In practice, an environment for modeling OML was created using Eclipse[3]. Eclipse is an open development platform that supports the creation of extensions. A set of tools shaped as Eclipse plug-ins provides the user with environment to specify and analyze the specification [49]. The recommended process

[3]For more info see www.eclipse.org.

for the creation of tools in this context is first to specify the tools using OML and then use the code generation tool from VDMTools and integrate this code with the existing Eclipse architecture. A parser and a type checker plug-ins for OML enable the user to model within Eclipse. The *showtrace* plug-in, a tool that has been developed for reading execution traces, displaying them graphically, aids in the analyzes of distributed real time systems [52]. In addition software to translate VDM++ specifications and proof obligations into HOL with the goal to automatically prove them [53]. This software will be presented more throughly in chapter 5. Also an annual workshop in which the people behind the Overture Community share their research on new topics and discuss the new trends and in which direction Overture should head [6].

1.3 Program Consistency

When programming, several kinds of errors can occur: syntactical or semantic errors. The first category are easily found by a parser of the language. The second is divided into two categories: the errors that can be detected statically and ones that only show up at 'run-time'.

Syntactical errors are normally perceived by parsers and do not pose great threat as typically a program will not compile or be interpreted if any syntactical error is detected. On the other hand, semantic errors depend of the selected concepts underlying each language. Strict type languages, for example, will ensure that values of a certain type can only be assigned to the variables of correct type. Finally some languages simply allow the user to do whatever he wants.

The semantics of a computer language can be seen as the meaning the sentences expressed in that language [45]. Semantics definition can have great value for programmers as it provides a precise definition of how the program will behave and lay ground for the analyzes and verification of it.

Static semantics defines if a syntactical correct sentence is well formed in a given set of context conditions for the language. Some properties of a language can be verified for consistency statically. This includes properties like: a operator being applied to the correct type operands or check for uniqueness of names when an object can be defined only once.

Dynamic semantics errors are errors that are difficult to be detected, even through testing that when appear normally result in a system breakdown [27]. Strongly typed languages, try to avoid this by imposing a strict type checking in which type mismatches are detected during the type checking.

So from the possibility of a program generating a error stems its inconsistency. If there is no chance it will generate errors the program is declared consistent. The notion of consistency in VDM models and what was done in VDM

to tackle these inconsistencies is explained in the following sections.

1.3.1 Consistency of VDM specifications

A VDM specification is inconsistency if there is the possibility of it generating an error. Otherwise it is consistent. The static semantics determines the consistency for all language constructs. The static semantics is implemented in a form of a type checker that checks the consistency of a specification, i.e. it verifies the well-formedness of all syntactical constructs. Other type of consistency is type consistency and the termination of recursive functions.

These other type of consistencies, like type consistency, is very difficult to verify in the type checking in VDM. A small example will illustrate why. In VDM it is possible to create union types. A union type corresponds to the set-theorectic union of two types, i.e. a type defined by a union contains all elements from both types that form the union.

```
foo : nat | bool -> nat
foo(x) == x + 1
```

The variable `x` belongs to a union type (`nat | bool`). So during evaluation, this expression is valid if `x` is in fact a `nat` or it will generate a error if it is a `bool`. So statically it is not possible to determine if this expression is consistent or inconsistent.

So indecidability stems from a rich type system making type checking undecidable in general. While other languages restrict the available facilities in order to get rid of some undecidable checks, the VDM approach is different. This approach will be described in the following sections.

When looking at VDM inconsistencies, several types can be distinguished like shown in figure 1.1. The outermost oval represents all the specifications that

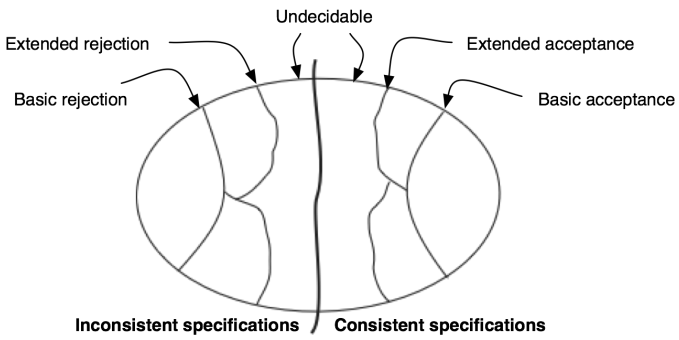

Figure 1.1: Types of specifications

can be written in a syntactically correct VDM form. The think line in the middle represents the border between inconsistent and consistent specifications. The

consistent ones are the ones that have at least one model in the semantics [34][4]. Within these two areas, further divisions can be made:

Basic: these are the models that can statically be accepted or rejected. This means that the type checker alone can determine their consistency or inconsistency.

Extended: the static check can give a definite answer on the consistency or inconsistency of these models. They are statically undecidable. A dynamic checker must be used to reason about the specification and determine them as being either consistent or inconsistent.

Undecidable: in these category fall the specifications that cannot be shown to be consistent or inconsistent by static or dynamic check. These specifications are left to the user for checking the consistency. So proof obligations are produced for the parts that cause the undecidability.

1.3.2 Possible and Definite mode

Two strategies exist for making the type check. The two strategies are:

Possible mode (POS): error messages indicate definitely inconsistent parts in the specification. The specification is rejected if some part is definitely inconsistent.

Definite mode (DEF): the specification is accepted if there is no errors. The specification is definitely consistent. However this strategy will fail for many VDM models because many constructs are possibly consistent but that cannot be decided in the static check.

1.3.3 Proof Obligations

The idea behind Proof Obligations (PO) is to obtain decidability of type checking by generating proof obligations for the parts that cause undecidability. A proof obligation of type consistency is an unproven theorem that must be proven in order to ensure the model to be internally consistent.

The purporse of this work is to generate PO but for termination of recursive functions. The idea is that if these PO are proved, it is certain that the recursive functions terminate, preventing this type of inconsistency.

A PO is a set of context assumptions and a predicate that must be proved based on these assumptions. A PO can also be regarded as a pointer to a piece of code that may generate 'run-time' errors, helping the work of the system specifier in this way.

[4] VDM dynamic semantics can be viewed as a functions that receives as input a VDM model and returns a set with all the semantically correct models. A model is inconsistent if this set is empty.

The POs are generated by the Proof Obligation Generator. The POG is invoked only if a specification is considered possible consistent by the type checker.

1.4 Theorem Provers

Mathematical proof of correctness of specific properties is typically a difficult task to accomplish. Some tools have been made to automate a part of this verification task and these are generally called theorem provers. In a theorem prover, the proof normally starts by entering a predicate that the user wish to be proved to be valid.

There are two ways to make the proof: interactive or automatically. In the interactive mode, the machine should be a helper while the user controls the proof direction. The proof system should facilitate the reading of the proof while making sure that the user guidances are correct [11]. In the automatic mode, the prover will try to prove a theorem by itself without requiring human interaction. However it is typically necessary to pre-define a tactic in order for the theorem prover to do its task. A tactic consist of a collection of logical rules and their order of appliance.

There are theorem provers based on different logics. In this thesis, the HOL theorem prover will be used in order to prove the termination POs. It is based on Higher Order Logic and will be extensively explained in section 2.1.

1.5 Recursive Functions

The term ”recursive function” is often used informally to describe any function that is defined with references to itself. Recursion is an algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task. Informally, a function defined by recursion is on that computes the result by ”calling itself”.

Every recursive solution involves two major parts or cases, the second part having three parts [38] .

- base case(s), in which the problem is simple enough to be solved directly, and
- recursive case(s). A recursive case has three components:
    1. divide the problem into one or more simpler or smaller parts of the problem,
    2. call the function (recursively) on each part, and
    3. combine the solutions of the parts into a solution for the problem.

1.5.1 Fixpoints and Least Fixpoints

The fixpoint of a function in general is a point of a function that it is mapped to itself.

$$x \text{ is fixpoint of } f \text{ if and only if } f(x) = x$$

The least fixpoint of f is the least element x that $f(x) = x$. In programming languages, recursive function definitions are formalized as computing a fix-point, typically the least fixpoint. A fixpoint of a high-order function f is another function p such that $f(p) = p$. A fixpoint operator is a function g that produces such fixpoint p for any function f:

$$f(g(f)) = g(f) \tag{1.1}$$

There is typically several functions that can satisfy this equation, the one that is normally used in computation is the least fixpoint.

Consider the factorial function:

$$\text{fact}(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1)$$

A single step of this recursive function can be defined as:

$$\mathbf{F} = \lambda f.\lambda x.\ \text{if } x = 0 \text{ then } 1 \text{ else } x * f(n-1)$$

Where the variable f will be the function *fact*. Applying the fixpoint operator to **F**:

$$\begin{aligned}
\textbf{fix}(\textbf{F})(n) &= \textbf{F}(\textbf{fix}(\textbf{F}))(x) \\
\textbf{fix}(\textbf{F})(n) &= \lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } (x * \textbf{fix}(\textbf{F})(x-1))(n) \\
\textbf{fix}(\textbf{F})(n) &= \text{if } n = 0 \text{ then } 1 \text{ else } n * \textbf{fix}(\textbf{F})(n-1) \\
\text{Abbreviating } \textbf{fix}(\textbf{F}) \text{ as } \textbf{fact}: & \\
\textbf{fact}(n) &= \text{if } n = 0 \text{ then } 1 \text{ else } n * \textbf{fact}(n-1)
\end{aligned}$$

So the fixpoint operator turns a non-recursive definition like **F** into a recursion function satisfying the equation 1.1.

1.6 Termination Proofs

Termination of algorithms is a central problem in software development. Formal methods for proving termination are essential in program verification [22, 21]. Though the termination problem is in general undecidable, several methods have been proposed for studying termination of functional programs [32]. In this thesis only the termination of recursive functions will be analyzed. The Proof Generating methods that matter to this thesis are the ones that can be fully automatic. Many theories have been formulated around this subject in the areas of term

rewrite systems, logic programs and functional programs [9, 32, 13, 21, 20]. The later are the more interesting for this thesis as the subject is termination of recursive functions (functional subset of VDM++).

To prove the termination of a functional program there has to be a well-founded ordering such that the arguments in each recursive call are smaller than the corresponding inputs. An ordering $\succ$ is well founded if there exists no infinite descending chain $x_1 \succ x_2 \succ \ldots$.

1.6.1 Orders

Total Orders

A total order is binary relation on some set X that is transitive, antisymmetric and total. If X is total ordered under $\preceq$ then the following properties are hold for all a, b and c in X.

- if $a \preceq b$ and $b \preceq a$ then $a = b$ (antisymmetric)
- if $a \preceq b$ and $b \preceq c$ then $a \preceq c$ (transitive)
- $a \preceq b$ or $b \preceq a$ (total)

A strict total order is somehow equivalent to a total order. The only difference lies on the use of the $\prec$ relation instead of $\preceq$. So the properties that hold are the same except for the first.

$$a \prec b \text{ if and only if } a \preceq b \text{ and } a \neq b$$

For termination proof there is the need to use a strict total order. Mainly because a non-strict total order leads always to infinite descending chains. For example, the next property is true in a non-strict order:

$$\ldots \leq 1 \leq 1 \leq 1 \leq 1 \leq \ldots$$

A totally ordered set (A,$\prec$) is said to have a well founded order if and only if every nonempty subset has a least element [10]. The total set of integers is an example of set that is not well founded because it has no least element.

Lexicographic Orders

A lexicographic order is a product of two orders. Given two partial ordered sets A and B, the lexicographic order is defined as:

$$(a, b) \leq (a', b') \text{ if and only if } a < a' \text{ or } (a = a' \text{ and } b \leq b')$$

One important property of lexicographic orders is that they preserve well-fondness if both sets are well the used sets are also well-founded with ($<$).

1.6.2 Measure

The measure function transforms the elements of the data type that is the argument of the function that termination should be proved. Since the arguments of functions can belong to any data type and many of them do not possess a well-founded order to relate them. So the measure is used to map the input data type to a data type that is well-ordered.

1.6.3 Estimation

Estimation is the process where the result of a function is estimated by its input and it is applied in the following way [21]. First for a function g the following *induction lemma* must be proved.

$$|x_i| \succeq |g(x_1, \ldots, x_n)| \tag{1.2}$$

Where $\succeq$ is a well-founded ordering and $|.|$ an arbitrary measure function.

If an equation of the following form is presented:

$$|t| \succ |\ldots g(r_1, \ldots, r_n) \ldots| \tag{1.3}$$

Then the instantiated version of lemma 1.2 can be used to *estimate* g by its i-th argument. The instantiated lemma is the following:

$$|r_i| \succeq |g(r_1, \ldots, r_n)| \tag{1.4}$$

Using 1.4 the equation, instead of proving 1.3, it is sufficient to prove:

$$|t| \succeq |\ldots r_i \ldots| \tag{1.5}$$

The transformation of 1.3 in 1.5 is called *estimation* of g *according to* induction lemma 1.2.

Such estimation of a function g within a term can only be applied if the function occurs at a position that is *monotonic*. This means that replacing g by something greater must increase the measure of the whole term.

1.7 Aims

The overall aim of formal methods is the development of correct programs. Specifying a model in a language like VDM-SL can help in this task. However the correctness of the initial specification cannot be proved. Although there are some forms of increasing the confidence that the model is correct like proof obligations.

One of the aims of this thesis is to upgrade the existing proof obligation generator of VDMTools to produce proof obligations for recursive functions. Since

the previous proof obligation generator does not consider this kind of inconsistency, this addiction can prove valuable to reason about the recursive functions. By proving the validity of these proof obligations the specifier can gain more confidence in the model, since it means that the recursive functions will not enter in a infinite loop, property that has much value specially when related with critical systems that must not fail.

The other aim is to clarify the limits of HOL on proving the above referred proof obligations. In addition inspect which ones can be proved automatically or user guided and provide tactics and strategies to make these proofs. In this way trying to continue the work that has been made previously in [53] that is to provide automatic discharge of VDM proof obligations.

1.8 Structure of thesis

An overview of what has been done in this thesis can be illustrated by figure 1.2 with the key steps numbered. In chapter 2 all the background necessary about

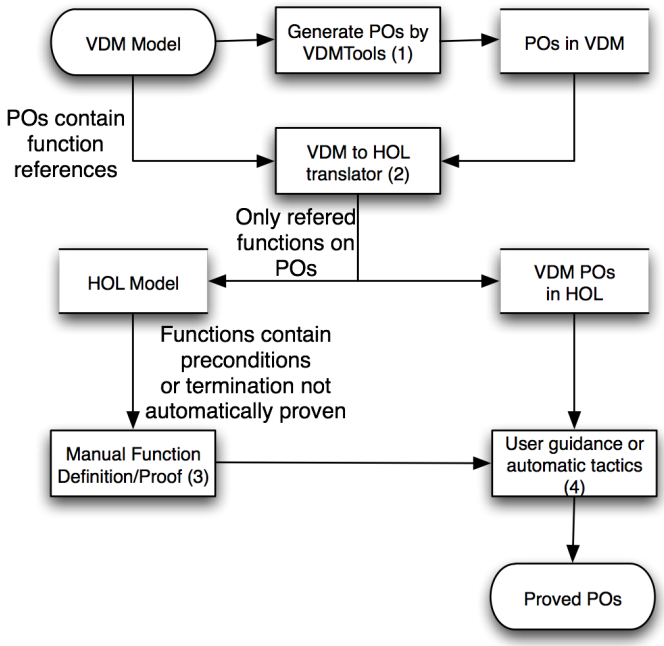

Figure 1.2: From VDM models to proved POs

VDM and HOL is provided. This background includes information on how to process step (3) from the figure. Readers that are familiar with both these topics can skip this chapter. In chapter 3 the theory behind the creation of termination POs is explained. Chapter 4 shows how this theory was implemented. Both these chapters are related with step (1) from figure. A introduction to VDM to

HOL translator (step (2)) and the changes made to it are described in chapter 5. A guide to how to make termination proofs in HOL is provided in chapter 6. This chapter is related with step (4). Finally, the concluding remarks are provided in chapter 7

1.8.1 Code snippets

Throughout the thesis, several pieces of source code in different notations are presented that are associated with different contexts. To make a better distinction they are surrounded with different style boxes.

For VDM specifications used as input object to this work, the box used is the following:

```
foo : nat -> nat
foo(x) = x;
```

Code snippets like this represent parts of VDM specifications that could be evaluated in VDMTools.

Alternatively, the next type of box indicates that the VDM code belongs to the actual specifications made in VDMTools. The code present in these boxes is highly simplified for better comprehension.

```
wf_Measure : FnBody -> bool
wf_Measure(fnb) == ...
```

The following box is used when something from the VDMTools interpreter is shown. Typically to show produced warnings or errors.

```
Warning[22]: ...
```

All the VDM boxes have rounded corners, the opposite of the HOL box presented next.

This box is used for the HOL interpreter:

```
- input
> output
```

Text in front of the '−' is the input for the interpreter while in front of '>' is the output of the inserted command.

Chapter 2

Background Information

In this chapter the background that is needed for HOL and VDM is presented. First an overview to HOL and its development and design. Then it is described the differences between automatic and manual proofs. Finally some HOL symbols, tactics and commands are explained. On the VDM part, a superficial analysis to the more important notions that are needed to understand this thesis are provided.

2.1 HOL

HOL is an automated proof system based on Higher Order Logic. It was originally created by the Automated Reasoning Group of the University of Cambridge with the purpose of making hardware verification [23].

The HOL System is an environment for interactive theorem proving in a higher-order logic. Its most important feature is its high degree of programmability through the meta-language ML. The system has a wide variety of uses from formalizing pure mathematics to verification of industrial hardware.

HOL4 is the latest version of the HOL automated proof system for higher order logic: a programming environment in which theorems can be proved and proof tools implemented [24]. Built-in decision procedures and theorem provers can automatically establish many simple theorems.

2.1.1 HOL development

In 1972, Robin Milner was making an effort to develop a proof-checking system called LCF (Logic for Computable Functions). It was designed to be a proof-checking program that allows the user to generate formal proofs about computable functions [35]. Since then, several descendants of LCF appeared,

in which HOL is included, and form now a prosperous paradigm in computer assisted reasoning [23].

A programming language ML (abbreviation of Meta Language, see [42]) was designed with strict type checking to ensure that values that could be created could be only acquired from axioms by applying a sequence of type inference rules. Though ML was first designed to support LCF it archived the status of a function programming language on its own. HOL is based on SML (Standard ML) and uses the MoscowML implementation [36].

The core system of HOL became stable in about 1988 using earlier versions of ML. During 1990's the HOL development split up into a open and a commercial version. The open version (called HOL90) utilizes the latest version of ML while the commercial (called ProofPower [48]) is based on the original version of ML. In 2004, an upgrade of HOL90 was release as HOL4 and it is the proof engine used in this thesis.

2.1.2 HOL Design

HOL is a system with the purpose of facilitating interactive and automated theorem proving. HOL consists of two layers: a meta-language layer and an object-layer. The HOL engines operates at the object-layer while commanding the HOL engine can be done through the meta-level. The latter is implemented in a ML language . There are two fundamental types in HOL that are worth mentioning:

term: An instance of this type represents an expression. It can have any type and does not need to be valid (e.q. $\forall(x : num)(y : num).(x > y \vee x < y)$).

hol_type: An instance of this type denotes the type of a term. In the above term it would be boolean, but many more types exist such as function types, lists, sets and user defined types

To express the difference more clearly, there is a *type_of* function that takes a term as input and returns a *hol_type*. Applying this function to the term above would result in 'boolean', being a *hol_type*. HOL makes use of many more types, but their presentation is not relevant for this thesis. Additional information can be found in the HOL4 description [25].

2.1.3 HOL proofs

As mentioned before, there is two ways to prove in HOL, automatic or user guided. In both cases it is necessary to tell the proof engine what to do. This can be expressed in *tactics*. A tactic tells the proof engine what proof steps to do.

The difference is that in an automatic proof, the tactic is supplied at the beginning and the user just has to see if the proof succeeded or not. In the user

guided mode, the user has to apply the tactics one at each time and conduct the proof until the goal is proved. There are a large number of built-in tactics and the base can be expanded with user made tactics. When an expression is proved to be valid, a theorem is produced. Theorems can be stored and used in other proofs.

In the user guided mode, a goal is introduced in the proof manager. Which is a tool to keep track of the proofs that are being made. During a proof, subgoals to the original goal might be originated by some proof steps and are automatically inserted to the proof manager. By solving all the subgoals, the original goals are consequently proven.

2.1.4 HOL symbols

HOL being a theorem prover, it is normal that the logic symbols are present. The following table shows what are the symbols in HOL and what they represent.

Name	Math. Logic	VDM	HOL
Implication	$A \Rightarrow B$	`A => B`	`A ==> B`
Negation	$\neg A$	`not(A)`	`~A`
Conjunction	$A \wedge B$	`A and B`	`A /\ B`
Disjunction	$A \vee B$	`A or B`	`A \/ B`
Truth	T	`true`	`T`
False	F	`false`	`F`
Forall	$\forall$	`forall`	`!`
Exists	$\exists$	`exists`	`?`
Infers	$A \vdash B$	n/a	`A |- B`

2.1.5 Proof Manager commands

In this section, the most frequently used proof manager commands in this thesis are introduced. Tutorials and guides can also be found in the internet [26, 37].

First, the essential command to start a proof. Without it nothing can be done. To start a proof the command to use is `g` (shorthand for goal). The command is invoked with an expression of type boolean that represents what is to be proved.

```
- g `!(x:num). x < x + 1`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !x. x < x + 1
     : proofs
```

By doing so, the expression is inserted into the proof manager becoming available for tactics to be employed.

The command to apply tactics is called `e` (shorthand for evaluation). This command is invoked with the tactic to apply.

```
- e( RW_TAC arith_ss [] )
OK..
> val it =
    Initial goal proved.
    |- !x. x < x + 1 : goalstack
```

The tactic employed, makes use of the set of arithmetic theorems to try to prove the goal. Simply by applying this tactic it was possible to prove the given boolean expression.

When an initial goal is proved, it is possible to store the theorem that it generates in a variable for later use. That can be archived with the command `top_thm()`.

```
- val THM1 = top_thm();
> val THM1 = |- !x. x < x + 1 : thm
```

Now that the theorem is stored in the variable *THM1*, it is possible to remove this proof from the proof manager with the command `drop()`.

```
- drop();
OK..
> val it = There are currently no proofs. : proofs
```

These are the basic commands to operate the proof manager.

2.1.6 Tactics

In this section the tactics that are used in this thesis are presented. Some of them are atomic and others can be called meta tactics because they can be put together with atomic tactics in order to obtain more complex ones. More information on these tactics can be obtained in [24].

RW_TAC it is a simplification tactic that provides conditional and contextual term rewriting. It has as arguments a simplification set and a list of theorems. The theorems are added to the simplification set and then it proceeds to simplify the goal. There are several built-in simplification sets: e.g. arithmetic, bool and lists. The most used one throughout the thesis is the arithmetic simplification set, that contain arithmetic theorems to simplify the goal. The list of theorems contains extra theorems to be rewrite like, for example, function definition theorems or other theorem that might be useful for the proof.

FULL_SIMP_TAC it is a simplification tactic that operates as the one above. The only difference is that this tactic also simplifies the assumptions and not just the goal.

STRIP_TAC it is used to remove one outermost occurrence of a connective (`!`, `==>`, `~`or `/\`) from the conclusion of the goal. It is often used in this thesis with the **REPEAT** tactic.

recInduct performs recursive induction with the supplied induction theorem.

WF_REL_TAC builds a tactic that starts a termination proof for HOL termination conditions given by *Hol_defn*.

DECIDE_TAC tries do decide if a boolean term is a tautology.

ASSUME_TAC adds an assumption to the goal. Normally it is used together with **Q.SPEC** rule, that is used to instantiate (or specialize) a theorem. The **Q.SPEC** rule is represented in a natural deduction style like this:

$$\frac{a : A.\ \forall y : A.\ P(x)}{P(a)}\ \forall\text{-E}$$

THEN it is a meta tactic that allows the construction of one tactic from two atom tactics. As the name indicates, it corresponds to apply the first tactic and then applying the second. It is a sort a tactic composition.

REPEAT it is a meta tactic that applies the argument tactic until it fails.

When using the rewrite or simplification tactic, it is possible to define how many times a theorem should be rewritten/simplified. This is extremely useful because if this number is not defined when applying this tactics with recursive functions, the result is infinite rewrites/simplifications. The number of times that the definitions is rewritten/simplified can chosen in the following way:

```
- RW_TAC simp_set [Once thm]
- FULL_SIMP_TAC simp_set [Ntimes thm n]
```

Where in the last one, `n` is a natural number.

2.1.7 Function definition in HOL

This section explains how to define functions in HOL. It might be needed for the proof of obligations in HOL. In order to use a definition of a function, its termination has to be proved, so the guidelines to prove termination (see chapter 6) can also be used in this section.

Define

The *Define* command allows the definition of functions in HOL. After using this command to define a function, HOL automatically tries to prove its termination. The function is only fully defined after the termination is proved. This command often fails to archive it automatically for recursive functions. HOL tries to generate a measure for the function itself and most of the times it is not successful, archiving only the most trivial proofs. This approach will never work with nested recursion [24].

If a function *foo* defined successfully using *Define*, the definition and induction theorem of the function are added to a HOL database under the name of *foo_def* and *foo_ind* respectively. To access these theorems the command *fetch* can be used.

Hol_defn

The *Hol_defn* command enables deferring the termination proof in the definition of recursive functions. This is, the proof of termination is not made at the definition time. Instead it creates a structure *Defn* which is a *Hol_type* that contains all the function information as well as the terminations conditions. This command is normally used when the '*Define* does not prove the termination automatically.

Some useful commands that are associated with *Defn* structure are shown below. These commands are all invoked in this form **Defn.command**.

eqns_of: extracts the function definitions from a *Defn* structure;

ind_of: extracts the induction theorems from a *Defn* structure;

set_reln: sets the relation to be used in the termination conditions;

tgoal: adds the termination conditions to the proof manager; and

tprove: if it successfully proves the terminations conditions using a provided tactic, returns the definition and induction theorem.

The tactic **WF_REL_TAC** is also associated with the *Defn* structure because it is used to its start the termination proof. So its use will be explained here. This tactic receives a term and starts a termination proof. This term is the measure that will be used in the termination proof.

So let `m` be a arbitrary measure function and let the termination conditions be in the proof manager.

```
- val foo_defn = Hol_defn "foo" `foo x = if x = 0 then 0 else foo(x-1)`;
> val it =
    HOL function definition (recursive)
```

```
    Equation(s) :
     [..] |- foo x = (if x = 0 then 0 else foo (x - 1))

    Induction :
     [..] |- !P. (!x. (~(x = 0) ==> P (x - 1)) ==> P x) ==> !v. P v

    Termination conditions :
      0. !x. ~(x = 0) ==> R (x - 1) x
      1. WF R : defn
- Defn.tgoal foo_defn
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ?R. WF R /\ !x. ~(x = 0) ==> R (x - 1) x
         : proofs
```

Now to actually start the proof the tactic needs to be applied.

```
- e (WF_REL_TAC `measure m`);
OK..
1 subgoal:
> val it =
    !x. ~(x = 0) ==> m (x - 1) < m x
    ...
```

Using the keyword *measure*, the relation is instantiated with ($<$) and the measure applied to both sides.

The other option is using the keyword *inv_image* that permits the use of a specified measure:

```
- e (WF_REL_TAC `inv_image $< m`);
OK..
1 subgoal:
> val it =
    !x. ~(x = 0) ==> m (x - 1) < m x
    ...
```

The result is the same if the relation ($<$) is used, but this one allows the use of lexicographic orders whereas in the other it is restricted to ($<$).

2.1.8 Lexicographic Orders

A lexicographic order can be used in HOL in this way:

```
($< LEX $<) (a,b) (c,d)
```

Of course the relations does not need to be ($<$) but in this thesis this is how it is going to be used.

2.2 VDM

In this section, a small introduction to VDM language is made. It is a small overview of all the features used throughout this thesis.

Throughout this thesis, only the functional subset of VDM is used. In VDM there is the possibility also to define operations. Operations have typically a state that is altered during the execution. So here, only the pure functional subset of VDM is considered.

2.2.1 Data Types

The notion of data types or simply types is central in VDM [17, 16]. The VDM types are more abstract than types from ”normal” programming languages. The VDM types fall into 2 categories:

- Basic types – simply store a value of the chosen type. Similar to other programming languages (e.g. integer, float, character)
- Constructed types – are formed out of applying a type constructor to either a basic type or a constructed type (e.g. maps, union types, composite)

New types can be declared in the following fashion:

```
NewType = (combination of existing types)
```

Basic Types

Basic types allow an atomic value to be stored like in any other programming languages [16]. A part of them should be familiar to any programmer as they are present in almost all programming languages. Basic types are: bool, integer, character, real, etc. Thus are the normal programming languages types and need no further explanation.

Constructed types

Constructed types allow the definition of more complex types based on the basic types and type operators. Constructed types that appear in this work are:

set: represents a set with no repetitions or order of any type. Associated operations are: **in** which is the typical set-theory belong; **subset** which given two sets checks if the first is subset of the second.

seq: represents a sequence or list. Associated operators are: the typical list operations head (**hd**) and tail (**tl**); **elems** which returns a set with the elements of the sequence.

map: used for a mapping from a unique key to a value. Associated operations are: **dom**, that returns the domain as a set; **rng** that returns the range of the map as a set.

compound type: can also be called record which is used to contain several variables of possible different types (called fields) in just on object. These fields can have an associated tags for easier manipulation.

Invariants

Invariants are properties registered in types to restrict their range of values.

```
Nat5 = nat
inv n = n >= 5;
```

This means that a variable of type `Nat5` can only hold a natural numbers greater or equal to 5. Invariants can be defined for any VDM type.

2.2.2 Logic in VDM

The relevant VDM logic symbols were already presented in 2.1.4. Here will be presented just an example of how a mathematical logic expression is translated to VDM.

$$\forall x \in \mathbb{N}.\ \neg(x = 0) \Rightarrow x > (x - 1)$$

Can be represented in VDM like:

```
forall x:nat & not(x=0) => x > (x-1)
```

2.2.3 Function definition in VDM

VDM has the normal construction blocks of a programming language and a functions can be declared like this:

```
foo : Type1 * ... * TypeN -> ResultType
foo(t1,...,tn) == ...;
```

Pre-conditions

The biggest difference from a ”typical” programming language is that functions can have pre-conditions, i.e. a predicate that must evaluate to true or the behavior of the function cannot be trusted.

```
foo : nat -> nat
foo(x) == x
pre x > 3;
```

This means that this function will only evaluate its argument (x) if it satisfies the pre-condition (x>3).

Using a pre-condition in a function definition will result in automatic creation of a ”pre-condition” predicate. This predicate has as name the prefix *pre_* and the name of the function. So for this example above, the precondition function would be:

```
pre_foo : nat -> bool
pre_foo(x) == x > 3;
```

Chapter 3

Generating Proof Obligations

This chapter explains the approach for generating proof obligations and the considerations that had to be made in order to get it working. First an introduction to the Proof Obligation Generator and POs is given. Then the concepts that were needed to introduce in VDM for termination POs are presented. Finally, through a series of examples it is shown the difference between the POs that have to be generated for different types of recursive functions.

3.1 Proof Obligations Generator

The current POG for VDM used in VDMTools was originally specified in [5]. The idea behind this approach is presented in [19, 12] and it was to extend the previous type checker by generating proof obligations for the parts that cause undecidability. The objective was to make the user aware of problematic parts in the model and also enable him to gain more confidence in the model by proving the POs.

The existing POG does not contemplate any generation of proof obligations for termination proofs. So the goal of this thesis is to extend this POG with the generation of these POs. These POs will make a specifier aware of possible termination inconsistencies in his/her models.

3.2 Proof Obligations

A Proof Obligation is a statement that must be proved in order to prove the consistency of some part of the model. The general form of a PO is that same of a proof rule. They contain the context information as assumptions and some property that must be proved as conclusion. In the next examples the POs will

be presented in an ASCII representation like this:

```
PO: context information => predicate
```

The POs are modeled in VDM as a proof rule with a list of hypotheses and a conclusion.

```
PrfRule :: hyp : seq of Sequent
           con : Expr
```

3.3 Ordering

As seen on section 1.6.1 proving termination involves finding a ordering and a measure in which the termination conditions can be proven correct. The ordering cannot be a random one, but it has to be a well-founded.

It has been decided that the terminations proofs present in this thesis will you use the ordering (Nat,$<$). In which $<$ is the typical relation of *"less then"* in natural numbers. Using this ordering prevents the task of proving that the chosen order is well founded. Mapping an input from a function to the Natural ordering is made by a measure function.

3.3.1 Lexicographic Orders

In some cases there is the need to use a lexicographic order (see section 1.6.1) in order to prove termination. The lexicographic order is used to compare tuples. As said above *nat* is well-ordered using '$<$' so if the sets chosen to form the order are both *nat* it will still be a well-founded order. Termination proof of some functions requires this kind of ordering.

In addition a lexicographic order can be used to compare any two arbitrary size tuples as long they have the same size. So if the measure function has a *nat* tuple as result, this order will be used for the proof obligation generation.

3.4 Measure

The measure is generally a function from input data type of the recursive function to a given well founded. As mentioned above the chosen order to make the termination proofs will be (Nat,$<$). Therefore all the measure functions in this thesis will have Nat as range. For a recursive function with this signature, a and b being generic types.

```
f : a -> b
```

The measure will have the following signature:

```
f_measure : a -> nat
```

Because of this decision, all the recursive functions that have input different from the *Nat* type will necessarily need a supplied measure function. For the recursive functions that have *Nat*as input the measure can be Nat_{id} by default but still other measure functions can be used.

The measure function is needed for the generation of all POs so it will be supplied for each recursive function definition using the keyword *measure*. The next example shows how this keyword is used in function f:

```
f_measure : a -> nat
f_measure(x) == ...;

f : a -> b
f(x) == ... f(x-1) ...
measure f_measure;
```

The keyword measure says that the function *f_measure* should be used as measure when generating the PO for termination of f.

3.4.1 Measure for Lexicographic Orders

If a lexicographic order is to be used in the termination proof of a function, the measure function has to be different then mentioned above. The range has to be a tuple of *Nat* as opposed to the single *Nat*. So whenever a measure function as a tuple as range, it means that in the PO a lexicographic order will be used.

```
meas_foo : a -> nat * nat
meas_foo(x) == ...;
```

Generalization for n-lexicographic orders

The lexicographic orders can be generalized to n-orderings so in VDM syntax that maybe indicated by a natural number after the *lex* keyword. In result of this the measure range has also to be a n-tuple of *nat*

```
foo : a * b * c * d -> e
foo(a,b,c,d) == ...
measure meas_foo;

meas_foo : a * b * c * d -> nat * nat * nat * nat
meas_foo(a,b,c,d) == mk_(...,...,...,...)
```

In this way to compare the tuples a lexicographic order of the product of the four values will be used.

3.4.2 VDM syntax change

So to enable the use of measure and for representing lexicographic orders in VDM some syntax changes had to be adopted. It is now possible to define the measure like shown in the sections above inside VDMTools.

A binary operator was chosen to represent lexicographic orders in VDM. The operator is an infix operator `(LEXn >)`. The `n` is the order of the order, they can be generalized to n-orderings like shown on the section right above. Although this operator is now defined, it is not possible to use it in a VDM specification and it will be only utilized when showing the PO. The use of the `LEX` operator is illustrated in the following example:

```
(A1,...,An) (LEXn >) (B1,...,Bn)
```

This means that the tuple on the left is greater then the one in the right, according to the lexicographic order.

3.5 Context generation

The context of the recursive application of a function is a conjunction of logical conditions that must hold to reach it during evaluation. A small example of how context can be constructed:

```
        if x = 0 then 1 else f(x-1)
```

The context for the *then* branch is that '$x = 0$'. Whereas for the *else* brach it is '$\neg(x = 0)$'.

3.6 Termination POs

The Termination POs of a recursive function are the conditions that must be proved in order the assure that it terminates. The POs contain a condition that should be proved for each recursive application of the function along withe their context as assumptions. The condition is that the measure of the recursive input should be smaller than the measure of the input in the ordering (Nat, $<$).

3.6.1 Simple Recursion

What is here called of simple recursion is a function with the following form:

$$f(x) = \ldots f(r) \ldots$$

And the factorial is an example of this type. It could be modeled in VDM in the following way.

```
id : nat -> nat
```

```
id(n) == n;

fac : nat -> nat
fac(n) == if n = 0 then 1 else n*fac(n-1)
measure id;
```

The recursive call is in the *else* branch so the context information for it is '$\neg(n = 0)$'. Then the statement that should be proved is created based on the argument of the function and the input for recursion. The PO for *fac* looks like:

```
forall n:nat & not(n=0) ==> id(n) > id(n-1)
```

Meaning that the recursive call argument should be smaller than the input of the function after the measure is applied on each of them. The PO contains as assumption the context of the recursive call as said above.

The process of generating POs for simple recursion is shown here in different steps:

1. Identify recursive functions;
2. Check if measure is defined; and
3. Check if measure has the correct type.

These analysis can be made by the type checker. If the type checks are passed the POs can finally be generated by:

4. Identifying recursive calls and their contexts; and
5. Producing the Proof Obligations with the collected data.

3.6.2 Nested Recursion

Two different types of nested recursion as distinguished bellow. The two ways of generating the POs are presented.

”Simple” Nested Recursion

One type of nested function is the one that in the recursion call uses a function different than the recursing one. It is called ”simple” in here because the process of generating the POs is simpler than the other type.

$$f(x) = ...f(g(r))...$$

The POs for this kind of recursion are generated the same way as the ones above for simple recursion as the termination condition is the same.

```
half : nat -> nat
half(n) ==
  if (n = 0) then 0
  else if (n = 1) then 0
  else 1 + half(n-2)
measure id;

log : nat -> nat
log(n) ==
  if (n = 0) then 0
  else if (n = 1) then 0
  else 1 + log(half(x-2) + 1)
measure id;
```

Since the *half* PO is of the same type of the one on the section above it will not be presented. The PO generated for *log* is the following:

```
forall n:nat & not(n=0) and not(n=1) => id(n) > id(1 + half(n-2))
```

So basically there is no difference from the POs in the section above.

Nested self recursion

The other type of nested recursion is when the nested function in the argument is the recursive function itself. The nested recursion can have arbitrary levels but here is the most simple case where the function has only one nested recursive call.

$$f(x) = ...f(f(r))... \tag{3.1}$$

This type of functions needs a different type of approach because the condition to assure termination is that argument of both calls to the function must be decreasing.

In the next example is shown how to generate POs for a recursive nested function called *nest* that is defined in VDM++

```
nest : nat -> nat
nest(n) == if n = 0 then 0 else nest(nest(n-1))
measure id;
```

The PO should contain conditions about both recursive calls of the function *nest*. The inner recursion yields the following PO.

```
forall n:nat & not(x=0) ==> id(n) > id(n-1)
```

For the outer recursion the PO is:

```
forall n:nat & not(x=0) ==> id(n) > id(nest(n-1))
```

These conditions state that the input for the recursive calls of nest should be decreasing for the innermost and outermost calls. So the final PO for *nest* is:

```
forall n:nat & not(x=0) ==> id(n) > id(n-1) and id(n) > id(nest(n-1))
```

3.6.3 Mutual Recursion

A mutual recursive function definition is characterized by two or more functions whose algorithms are mutually dependent. This is, one function f to calculate its result calls another function g, but again to find its result, g needs to call f creating a mutual dependency.

$$f(n) = ...g(n')...$$
$$g(r) = ...f(r')...$$

POs for mutual recursive definitions only differ from the ones above because they involve a measure from each function present in the definition. Apart from that the process is the same. The functions presented below are mutual recursive. The measure for them is the same (Id) but here they will have the function name attached to the measure for better comprehension.

```
even: nat -> nat
even(x) ==
  if x = 0 then true
  else odd(x-1);
measure id_even;

odd : nat -> nat
odd(x) ==
  if x = 0 then false
  else even(x-1)
measure id_odd;
```

The recursion in *even* yields the right side of the PO conjunction and *odd* the left side. They says that the input for the recursive call of *odd* (*even*) should be smaller then the input of the *even* (*odd*) respectively.

```
forall x:nat & not(x=0) => id_even(x) > id_odd(x-1)
and
forall x:nat & not(x=0) => id_odd(x) > id_even(x-1)
```

If the arguments are decreasing in each of the functions then the mutual recursive definition can be said as terminating.

3.7 Preconditions

Proof obligations for functions with precondition require that it is recorded as an assumption. The relation between input and recursive argument must be valid only if they fulfill the precondition. The next function *fnpre* illustrates this:

```
fnpre : nat -> nat
fnpre(x) ==
  if(x=3) then 3
  else fnpre(x-1)
pre x >= 3
measure id;
```

For this function a predicate named '*pre_fnpre*' is created that computes its precondition. This predicate will be used on the construction of the PO. The PO for *fnpre* is:

```
forall x:nat & (pre_fnpre(x) => not(x=3)) => id(x) > id(x-1)
```

It is only required that the recursion relation is valid in the values defined for the precondition. So functions with preconditions can be treated as every other, they just have the precondition added to the context of each recursive case.

3.8 Using Lexicographic Orders

Some recursive functions require a different approach using a lexicographic order (see section 3.3.1). The next example which is the breath-first traversal of graph need this type of approach or the generated POs would be unprovable.

```
types
Graph = map nat to seq of nat
inv g == forall i in set dom g & elems g(i) subset dom g;

functions
depthf : seq of nat * Graph * seq of nat -> seq of nat
depthf(l,g,vis) ==
  cases l:
  [] -> reverse vis,
  h^[t] -> if h in set vis
              then depthf(t,g,vis)
              else depthf(g(h)^t,g,h^vis)
pre elems l subset dom g;
```

The first instinct is to use the measure on the parameter 'l' in which the recursion is made.

```
meas_depthf1 : seq of nat * Graph * seq of nat -> nat
meas_depthf1(l,g,vis) == len l;
```

But as seen in the generated POs below this is not enough to prove termination. The measure function has been rewritten to its definition for better clarity.

```
forall l:nat, g:Graph, vis:seq of nat &
  not(l=[]) and h in set vis and t = tl l
    => meas_dephf1(l,g,vis) > meas_dephf1(t,g,vis)
```

```
  and
  not(l=[]) and and not(h in set vis t) and t = tl l and h = hd l
    => meas_dephf1(l,g,vis) > meas_dephf1(g(h)^t,g,h^vis)
```

The second part of the conjunction would only be true if the graph g had not links. So with this measure is impossible to prove the termination. One can try with another measure but this one related with the visited nodes of the graph.

```
meas_depthf2 : seq of nat * Graph * seq of nat -> nat
meas_depthf2(l,g,vis) == card dom g - len vis;
```

The generated POs with the measure are presented below:

```
forall l:nat, g:Graph, vis:seq of nat &
  not(l=[]) and h in set vis and t = tl l =>
   meas_depthf2(l,g,vis) > meas_depthf2(t,g,vis)
  and
  not(l=[]) and not(h in set vis t) and t = tl l and h = hd l =>
    meas_depthf2(l,g,vis) > meas_depthf2(g(h)^t,g,h^vis)
```

But the first part of this conjunction is false. And therefore is impossible to prove termination in this way. Also combining the two measure with a sum is not enough.

```
meas_depthf3 : seq of nat * Graph * seq of nat -> nat
meas_depthf3(l,g,vis) == meas_depthf1(l,g,vis) + meas_depthf2(l,g,vis);
```

Also the second condition is not respected using this measure.

So in order prove the termination of this function, a lexicographic order must be used. The definition of *depthf* but if the *measure* function has as range a tuple of *nat* a lexicographic order will be used to do the comparing.

```
depthf : seq of nat * Graph * seq of nat -> seq of nat
depthf(l,g,vis) == ...
measure meas_depthf;

meas_depthf : seq of nat * Graph * seq of nat -> seq of nat -> nat * nat
meas_depthf(l,g,vis) ==
  mk_(card dom graph - len vis, len l)
```

The measure in the case of a lexicographic order the measure used has to have a pair of *nat* as range. And the PO will need to have the symbol *lex* to indicate it is an lexicographic order.

```
PO : forall l:seq of nat, g:Graph, vis:seq of nat &
         not(l = []) and h in set vis and t = tl l =>
           meas_depthf(l,g,vis) (LEX2 >)  meas_dephf(t,g,vis)
         and
         not(l=[]) and not(h in set vis t) and t = tl l and h = hd l =>
           meas_depthf(l,g,vis) (LEX2 >) meas_depthf(g(h)^t,g,h^vis)
```

Chapter 4

Updating VDMTools Type Checker and POG

This chapter is about the changes needed to be made to the VDMTools type checker and POG in order to generate the Termination Proof Obligation. First an overview of the whole process is given and then it is shown how the type checker and the POG models have been updated. Then the updates made in the static checker are explained, including how recursion is found, warnings and errors that are produced. Furthermore the alterations to the POG are explained, showing how the POs are created based on the information collected in the static check phase.

4.1 Previous Type Checker and POG

The previous existing type checker and POG had more then 50.000 lines of VDM-SL all together and a characteristic way of work. Updating them involves getting to know how the mechanism for type checking and generation of PO works. This mechanism had to be carefully studied so that the changes do not impact the previously working code in any unforeseen way. To ensure this, after all the changes were done, a testing battery was ran over the new specification and the results were compared with the results obtained previously to the alteration.

Figure 4.1 illustrates the data flow from the parser until the POs are formed. The parser extracts the abstract syntax tree from a VDM model. This tree is passed on to the type checker. The type checker rejects the specification if it is definitely inconsistent, otherwise the specification can be given to the proof obligation generator. The POG will then produce the POs and display it in the GUI of VDMTools or dump them so they can be used in an external proof tool.

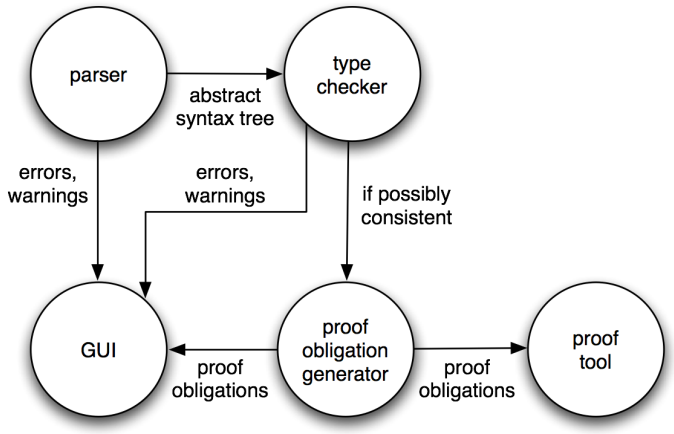

Figure 4.1: From the Parser to POs

The changes presented in this chapter where made in type checker and in the POG. The process of generating the POs can be distinguished in two phases. The type checking phase which is the tasks that are made by the type checker and the POG phase in which the POs are generated. The POG phase is dependent on the type checking. Only if the type checker accepts the specification is possibly correct the POG is invoked.

4.2 Type Checker - Static Semantics

The VDM syntax was altered as according to section 3.4.2. The *measure* should have the same data type as the input for the function and *nat* as result. So when the *m*easure keyword is inserted in one function, the type of the *measure* should be inspected.

The type checker task is to reject incorrect *measure* definitions. Because in the altered VDM++ syntax the *measure* can be inserted for each function. The *measure* type checking is made even if the function is not recursive. The rejection does not imply the total halt of the the termination proof generation. If an error is detected that makes the generation of a proof obligation impossible, this error will be reported as a warning. The reason is that if a user is interested in get the termination obligations he should look at the warnings and fix what is needed. But if he is not interested in it he might as well ignore the warnings.

It is in the type checker that the main part of the work is carried out. The type checker should calculate which functions are recursive and mutual recursive and make all the tests that ensure that the PO can be generated. This process is explained in detail in the subsection below.

The alterations were made at top level of the type checker, i.e. all the pro-

cesses explained below are made at a class level while inspecting a specification. It is at class level that the well foundness of the class is decided although some errors can come from lower levels.

The static semantics of a specification is checked one class/module at a time. This means some of the termination related checks were impossible to be made if the functions in the class/module involved recursion with functions from other classes/modules. Thus some changes had to be made in order to facilitate these checks.

4.2.1 Creating the recursion map

The recursion map is the structure that contains the information of what identifiers present in a function lead to recursion. This information will be stored in the environment. The environment is accessible to the type checker and the POG and it is the only medium through where they can communicate. So the environment in the existing definitions for VDMTools had to be changed in order to accommodate information about recursion.

Version Control Manager alterations

The Version Control Manager (VCM) is a VDM-SL module responsible to hold the data that is derived by the parser. Identifiers used, types of functions parsed and whatever is needed for the type checker to be able to type check a specification. The VCM structures are exported during the parsing of the specification, this is, before the type checking. To detect recursive functions, a new field had to be added to the VCM state. This field will contains the definition of all functions present in the specification. The definitions are needed to construct the recursion map, because the type checking is made over a single class at one time. This means that if a function has calls to functions in other classes the detection of the recursion would not be possible if there was no way to access it. Furthermore, there is the need for the whole functions definition because these definitions have to be searched for identifiers in the function bodies, not just its type as it was before the alterations. So a map with all functions with their name as keys was added to the *ParseTypeInfo*. For each class there is a structure of these types containing several informations about the class.

```
ParseTypeInfo ::
  nm    : AS`Name
  ...
  fndefs : map AS`Name to AS`FnDef
```

The field added contains the functions defined in each class. This will enable the possibility of investigate a class anywhere in the type checker by accessing the VCM.

Environment alterations

A data type to store all the information about the functions needed for Termination POs was defined and added to the environment. It is a map that has as keys function names that are related with their information. This is, for each function, a set with functions called by it, it's measure if it exists and a boolean that indicates if a PO shall be printed for this function.

```
FunctionInfo ::
  rec   : set of AS`Name
  meas  : [AS`Name]
  printPO: bool;

state TypeCheckVals of
  ...
  RecMap        : map AS`Name to FunctionInfo
```

During the recursion calculation, this structure will be used for two different purposes:

- at first, as explained in the section below, it is used to keep all the identifiers of function calls present in each function body. These called functions might not lead to recursion, they will just be used as input for the recursive function detection algorithm; and
- then it will be used to keep the names of the definitely recursive functions, that are found through the application an algorithm described in a section below called Detecting Recursion. It will be saved in this form, it will be saved in the environment and then being used in the POG.

Recursion Map: first stage

As said above the recursion map has two stages, firstly all the functions with the functions that they call are put into the map.

The type check is constructed class by class, the recursion map is also made this way. So to make the recursion map for a class, it is necessary that all functions in a class are inserted into the map. However this is not sufficient, as a class may contain references to functions in other classes, implying that these also need to be investigated. The process is the following:

1. For each function in a class find the set of identifiers that are used in its body.
2. Not all these identifiers are functions so after getting them they should be intersected with the set of all functions in order to retain only the function applications.

3. After having all functions and their called identifiers, these should be inserted in the map.

4. If after this, some function identifier is not present in the domain but it is present on the range, it means that this function was still not inspected. The identifiers present in these functions need to be searched in process similar to step 2).

5. Repeat step 4) until all functions present in the range of the map are also in the domain.

After all this, all the functions and their called functions that are needed to calculate the recursion in a certain class are present in this map. And this map can be processed in the way shown below in order to find the recursive functions.

Recursion Map: Detecting Recursion

Now with the first version of the recursion map, the algorithm to find the recursion can be applied. Important to say is that just because a function is present in this version of the recursion map does not mean it has necessarily recursion. This algorithm transforms a raw recursion map, with the function and all the called functions, in the recursion map with only the functions and the called functions that lead to recursion.

With this map it is possible to identify which functions are recursive or not. The strategy is to follow the trace of each of the recursive calls.

```
f(x) == g(x')...h(x')...k(x')...;
g(x) == ...h(x')...;
h(x) == ...g(x')... k(x')...;
k(x) == ...;
m(x) == ...m(x')...;
```

The resulting map for this case is (information not relevant for detecting recursion is not displayed):

```
{
 f |-> {g,h,k},
 g |-> {h},
 h |-> {g,k},
 k |-> {},
 m |-> {m}
}
```

This map is then searched for functions that definitely do not have recursion. In this case *k*, because it calls no function (`k |-> {}`). If a functions is detected definitely non recursive it is also removed from the set of called functions in the

range of the map. This search is repeated until no changes are detected after it is applied.

```
{
 f |-> {g,h},
 g |-> {h},
 h |-> {g},
 m |-> {m}
}
```

After the cleaning of the non recursive functions, the recursive traces of the function calls must be found. This data type is then transformed into one that allows the trace search. The data structure used for calculating this traces is:

```
RecMapTemp = map Name to set of (seq of Name);
```

The sequence allows to keep the trace during the algorithm. A kind of transitive closure is then applied to this structure. In the section below it is explained how this transitive closure works.

```
{
 f |-> {[g],[h]},
 g |-> {[h]},
 h |-> {[g]},
 m |-> {[m]},
}
```

Finding the trace involves repeatedly inspecting the last function called in a sequence and add the called functions by it to the sequence. The tracing stops if the functions to be added to the sequence are already present there.

```
{
 f |-> {[g,h],[h,g]},
 g |-> {[h,g]},
 h |-> {[g,h]},
 m |-> {[m]},
}
```

Now the recursion can be identified. The function names that have a sequence that finishes with their name are defined used recursion. If the sequence has only one element then the functions is directly recursive. Otherwise it is defined by mutual recursion and the function call that leads to recursion is the first element in the trace. If a trace does not finish with the same name as the function pointing to it then this function calls recursive functions but it is not recursive itself. After finding which functions are recursive and what calls lead to the recursion the context map is altered. It will contain only the effective functions that lead to recursion and not all the functions that are called.

```
{
 f |-> {},
```

```
 g |-> {h},
 h |-> {g},
 m |-> {m},
}
```

This is now in its final. Here all identifiers which are present lead to recursion. Another round of cleaning the map can then take place, leaving only the functions that effectively have recursion.

Transitive Closure

The idea of this algorithm is to apply a transitive closure, but also keep track of the path is being followed while applying it. It is shown here how to make a single step of this algorithm.

```
RecMapTemp = map Name to set of (seq of Name);

TransitiveStep : RecMapTemp * RecMapTemp -> RecMapTemp
TransitiveStep(m,n) ==
  { x |->
    { if last(y) in set dom n
      then
        dunion
          { if {hd x} inter elems trace = {}
            then trace^[hd x]
            else trace | x in set fnm(key)
           }
      else y
    | y in set m(x)}
  | x in set m}
```

The transitive closure is then obtained from applying this step until the output is the same as the input.

```
TransitiveClosure : RecMapTemp -> RecMapTemp
TransitiveClosure(m) ==
  let mTrans = TransitiveStep(m,m)
  in
    if m = mTrans
    then m
    else TransitiveClosure(mTrans);
```

4.2.2 Warnings and Errors

At this point, the recursion map has been built as shown in the subsection above. So it is possible to generate the warnings and errors related with recursion.

The style chosen was to issue errors when there is a sign that the user wants to really generate the termination POs. This is, if a measure is defined and there

is some inconsistency in it, a error will be reported. Otherwise, warnings will be issued so that the user is aware.

All warnings and errors are presented in the following subsections.

Function is recursive but does not have measure

If a function is detected as recursive but does not have a defined measure a warning is issued. When inspecting a function definition for static errors, if the function name is in the recursion map, it means it must have a measure.

```
foo : nat -> nat
foo(n) == if n=0 then 0 else foo(n-1);
```

The issued warning is the following:

```
Warning[500] : foo is recursive but does not have a measure defined
```

Function and its measure do not have the same domain

As mentioned in section 3.4, the measure must have the same domain as the function. If the domain of both does not match then a error will be issued and not PO generated.

```
foo : nat -> nat
foo(n) ==
  if n = 0
  then 0
  else foo(n-1)
measure idf;

idf : int -> nat
idf(x) = abs(x);
```

The warning is the following:

```
Error[501] : foo and its measure do not have the same domain
```

Measure range is not nat or a tuple of nat

Also mentioned in section 3.4 is that the allowed ranges for measures are *nat* or a tuple of *nat*. So if a measure has a different range from these a error is issued and the PO will not be generated.

```
foo : nat -> nat
foo(n) ==
  if n = 0
  then 0
  else foo(n-1)
```

```
measure idf;

idf : nat -> real
idf(x) = x;
```

And the warning issued is the following:

```
Error[502] : foo measure range is not nat or a tuple of nat
```

Funtion is mutual recursive with an other and the last does not have measure defined

As shown in section 3.6.3, mutual recursive functions need to have a measure for each one. A warning is issued if the measure for the functions that lead to the mutual recursion is not found.

```
id_even : nat -> nat
id_even(x) == x;

even: nat -> nat
even(x) ==
  if x = 0 then true
  else odd(x-1);
measure id_even;

odd : nat -> nat
odd(x) ==
  if x = 0 then false
  else even(x-1);
```

The warning issued is the following:

```
Warning[503] : even is mutual recursive with odd
               and odd does not have measure defined
```

Measures of mutual recursive functions do not have the same range

If the result of two measure results need to be compared, they need to have the same type. So if it is detected an inconsistency in the measures that are need for a PO, it will not be generated and a warning will be issued.

```
id_even : nat -> nat
id_even(x) == x;

even: nat -> nat
even(x) ==
  if x = 0 then true
  else odd(x-1);
measure id_even;
```

```
m_odd : nat -> nat * nat
m_odd(x) == mk_(x,x);

odd : nat -> nat
odd(x) ==
  if x = 0 then false
  else even(x-1)
measure m_odd;
```

The warning is the following:

```
"Warning[504] : The measures of even and odd must to have the same range"
```

Non recursive function with a measure defined

This warning does not have any influence on the PO generation, but if a non recursive function does not have a measure a warning is issued saying the function does not need a measure.

```
id : nat -> nat
id(x) == x;

foo : nat -> nat
foo(x) == 3
measure id;
```

The warning is the following:

```
"Warning[505] : foo has a measure but it is not recursive"
```

Measure identifier does not exist

This error is not connected to the proof obligations but if a unknown identifier is found as measure it should be reported as an error. This is the only new error that the static checker generates.

```
id : nat -> nat
id(x) == x;

foo : nat -> nat
foo(x) == if x = 0 then 0 else foo(x-1)
measure ID;
```

The warning is the following:

```
"Error[34] : Unknown identifier ID"
```

4.2.3 Updating the environment

The final recursion map that contains information about the recursive functions and for which ones should the POs be printed, is then added to the environment. This is the only way to channel the information to the POG. This information is built in each class at one time and then appended to the recursion map. So the final map is updated several times. In the beginning of the static check for a class, the recursion map is obtained from the environment and then updated with the inspected functions on that class.

4.3 Proof Obligation Generator - Dynamic Semantics

The main work is made by the POG that generates the POs that ensures consistency. The POs are generated by traversing the VDM AST recursively. As the POG goes down the tree the context is constructed incrementally constructed incrementally. The POs are generated on detection of a VDM construct that must be checked for consistency. The POG will only be invoked if the Type Checking is successfully passed for possible type correctness. So in this phase it is assured that the *measure* for the functions are type correct if the field *printPO* indicates so (this field belongs to the structure presented in page 38).

4.3.1 Context

As mentioned above, as the POG goes through the abstract syntax tree it constructs the context. In the context, things like the arguments of the function, what variables are being used, and properties about variables are collected. This information is available and it is used at the time of PO generation.

4.3.2 Generating the POs

So at this point all that is needed for the generation of the POs is already gathered in the context. The only thing needed is to go down through the abstract tree in VDM and find the apply expressions. Then the recursive function information map must be consulted in order to find out if the function in which the expression appears is recursive and if the name of the expression leads to recursion. If nothing is found then the expression does not requires a PO. On the other hand, a PO is generated if in the function information there is the indication that this apply expression name leads to recursion.

```
VerifyFuncRecursion : Name * seq of Expr * Context -> seq of ProofObligation
VerifyFuncRecursion(nm,arg,ctxt)==
  let recMap = ctxt.recmap,
      curfunc = ExtractFnm(ctxt.loc)
  in
```

```
  if curfunc in set dom recMap
  then
    if nm in set recMap(curfunc)
      then
      if recMap.(nm).printPO
      then
      GenerateTPO(nm,recinfo(curfunc).meas,recinfo(nm),arg,ctxt)
      else []
    else []
  else [];
```

As mentioned above, the PO shall be printed if that is possible, which is recorded in the *printPO* field. The function *GenerateTPO* is responsible for creating the condition that must hold in the PO and also collect the environment.

```
GenerateTPO : ... -> BinaryExpr
GenerateTPO(nm,m1,m2,arg,ctxt) ==
  if m1 <> m2
  then
    let m1tprng = extractRange(m1),
      params = extractParamsFromCtxt(ctxt)
    in
      cases m1tprng:
        mk_ProductType(ltp) ->
          mk_BinaryExpr(
            mk_ApplyExpr(m1,parms),
            mk_(<LEXORD>, len ltp),
            mk_ApplyExpr(m2,arg)
          ),
        others -> mk_BinaryExpr(
            mk_ApplyExpr(m1,parms),
            <NUMGT>,
            mk_ApplyExpr(m2,arg)
  else ...
```

The missing *else* part above is almost identical, the only difference is that the same measure is used for both. The expression that is generated depends on the range of the measures. If it is a product type it means a lexicographic order ($<$*LEXORD*$>$) is needed to compare them. Otherwise $>$ ($<$*NUMGT*$>$) will be used. The parameters of the function can be extracted from the context. This is all that takes to generate the POs. All the major work was done in the static check so the POG solely prints the POs.

The code presented just generates the PO conclusion since functions to inspect the context and build the PO assumption. This function receives as input the context and the conclusion and returns

Chapter 5

VDM to HOL translator

The VDM to HOL translator is part of an effort to bring automatic proof closer to VDM [53]. It consists of VDM to HOL translator and a bunch of HOL tactics that help in the proof of the POs. The main idea is to transport VDM models and their POs into the HOL language and then reason about the POs directly in HOL.

Some problems might appear from the use of the translator beside the not complete coverage of the VDM language . These are presented in this chapter. In addition some updates were made to prevent some of the problems and they are also presented in this chapter.

This tool was used to make all possible translations that appear in the next chapter.

5.1 Pre-conditions

The existing *VDM to HOL* translator does not produce the correct functions definitions if they contain pre-conditions . The problems is that a function and its pre-condition are translated into two different functions in HOL with no connection whatsoever. Take for example the following function in VDM:

```
foo : nat -> nat
foo(x) ==
  if x = 1 then 1
  else foo(x-1)
pre x > 0;
```

Using the *VDM to HOL* translator, two functions are created:

```
Define
  `foo (x:num) = if (x  = 1) then 1 else (foo (x  - 1 ))`;
Define `pre_foo (foo_parameter_1:num)
  = (let x = foo_parameter_1 in (x  > 0 ))`;
```

Although the function in VDM is clearly terminating. The termination of the one that results from the translation is impossible to prove because the termination condition that HOL would generate for this function would be:

```
!(x:num). ~(x=1) ==> x > x - 1
```

Which is clearly not valid if $x = 0$. To solve this problem, the function *foo* had to be manually defined by introducing the PO directly in a HOL function definition.

```
Define
  `foo (x:num) =
    if pre_foo(x) then
      if (x  = 1) then 1 else (foo (x  - 1 ))
    else ARB`;
```

However this is not yet enough to enable HOL to prove *foo* definition automatically, because the definition of *pre_foo* and a "let" expression related theorem have to be added to HOLs automatic termination proof tactic. So for each pre-condition translated, a command to add it to this tactic is also generated. The result of *foo* translation is now this:

```
Define `pre_foo (foo_parameter_1:num)
  = (let x = foo_parameter_1 in (x  > 0 ))`;
BasicProvers.export_rewrites(["pre_foo_def"]);
TotalDefn.default_termination_simps :=
  (fetch "-" "pre_foo_def") :: !TotalDefn.default_termination_simps;
Define `foo (foo_parameter_1:num)
  = (let x = foo_parameter_1 in
    (if (pre_foo foo_parameter_1)
     then (if (x  = 1 ) then 1
     else (foo (x  - 1 ))) else ARB)
    )`;
BasicProvers.export_rewrites(["foo_def"]);
```

The variable `default_termination_simps` holds a list of theorems that are used when HOL tries to automatically prove termination. The pre-condition definition is being add to this list.

As mentioned before, the proof is made with a bunch of auxiliary tactics that were defined by the author of the translator. These tactics are present in a file that should be loaded into HOL before the proof takes place. In this file, the "let" expression theorem must be also be added to the automatic termination tactic:

```
TotalDefn.default_termination_simps
  :=  boolTheory.LET_THM :: !TotalDefn.default_termination_simps;
```

5.2 Function Definitions

Both of the problems presented in this section could be solved if the recursive map structure could be exported from VDMTools.

5.2.1 Recursive Functions

As seen in the example above, the translator generates a line for all functions that adds their definitions to the list of rewrites to be used in the VDM to HOL tactics:

```
BasicProvers.export_rewrites(["function_def"]);
```

Doing so with recursive functions will lead to infinite rewrites. The reason for this is explained on section 2.1.6.

5.2.2 Mutual recursive functions

Mutual recursive function in HOL must be declared in the same define block. Translating mutual recursive function using this translator will originate two define blocks, one for each function and an error indicating this. These blocks will not be accepted by HOL. This is the error produced by VDM to HOL when it is used to translate mutual recursive functions.

```
Error, the HolAst cannot be printed, not all dependencies can be satisfied.
Most likely due to mutual recursive definitions
```

Chapter 6

Discharging Proof Obligations in HOL

This chapter is centered on ways to discharge the POs for Termination of recursively defined functions. Several ways of doing this will be presented: discharging them using HOL in a automatic or user guided strategy.

Throughout the chapter, examples are shown in order to illustrate the different approaches to elaborate the proofs.

Most of the proofs shown here, appear to be realized in one single step. However they had to be made first in the interactive mode to find out which are the necessary tactics.

6.1 Termination Proofs in HOL

There are two ways of generating this conditions: using the POG from VDMTools and then translate them or using HOL directly. The two options are very similar. The second way seems to make the first one redundant, but the POs generated by VDMTools were not designed specifically to be proved in HOL.

6.1.1 From VDM POs to HOL POs

From a VDM model, one can generate the termination POs with VDMTools. If the generated POs contain references to any VDM model function, it means that these functions will also need to be translated to HOL in order to give meaning to those references in HOL.

When HOL is used to prove properties about a VDM model, it is necessary to define semantically equivalent functions in HOL and they need to be proved to terminate. This essentially leads to redundant termination proofs because the

POG for VDM also generates the same termination POs. In some cases HOL is able to prove automatically, but just the simplest cases, so function definition in HOL might involve manual proofs (see section 6.1.2).

When all dependencies are solved, the user may apply the tactics that try to prove automatically or try to prove the POs himself in the case they cannot be proven automatically.

6.1.2 Differences in generated POs from VDMTools and HOL

The major difference between these two methods is that POs generated by HOL are more general. This is, they allow that a custom relation to be selected when attempting a proof. Whereas the VDMTools POs are restricted to the *less then* ($<$) relation from the natural numbers. This option has been already explained in section 3.3.

Inspecting the factorial example, the difference can be explained.

```
id : nat -> nat
id(n) == n;

fac : nat -> nat
fac(n) == if n = 0 then 1 else n*fac(n-1)
measure id;
```

The generated PO would be this as seen in previous chapters.

```
forall n:nat & not(n=0) ==> id(n) > id(n-1)
```

Using the *VDM to HOL* translator the result would be:

```
(!  uni_0_var_1.
((((inv_num uni_0_var_1)  /\ (?  n.(n  = uni_0_var_1 ))  )  /\ T )  ==>
 (let n = uni_0_var_1 in
 ((~ (n  = 0 )) ==> ((id n)  > (id (n  - 1 )) ) )) ))
```

Apart of looking terribly complex, it is just a way to make sure that the correct invariants are being taken into account. Excluding that detail, the PO looks exactly the same as before. The similarities can be seen if just the first and last line are considered. In the remaining part of this thesis , the POs presented will be in its simplified style for the sake of readability unless it is stated otherwise. They will not including invariant information when not needed and they will have the *let* expressions already substituted.

To obtain the HOL counterpart, the command *Hol_defn* must be used (this command is explained in further detail in section **??**). Using this command requires the function to be translated to HOL. This can also be generally archived by the *VDM to HOL* translator. The *VDM to HOL* translator will always try to define the function with the automatic proving command '*Define*' which will fail

in many occasions for recursive functions. It would not fail in following example but it is only used to illustrate the use of the command '*Hol_defn*'.

```
- val fac_defn =
  Hol_defn "fac"  `fac(n) = if n = 0 then 1 else n * fac(n-1)`;
> ...
  Termination conditions :
      0. !n. ~(n = 0) ==> R (n - 1) n
      1. WF R : defn
```

This PO or termination condition, as they are called in HOL seems more complex, mostly due to the two clauses, but the only difference is that **R** is a general relation that must be well founded according to condition 2. Where in the PO generated by VDMTools, **R** is already instantiated with $(<)$ which is well founded thus the non existence of a well founded clause. An additional difference is the measure. It is not present in the shown termination conditions, but can be added later when the proof is actually initiated.

To turn the condition to exactly the same as the generated PO one just has to instantiate **R** with $(<)$ by doing the following command:

```
- Defn.set_reln fac_defn ``(<)``;
> ...
  Termination conditions :
      0. !x. ~(x = 0) ==> x - 1 < x
      1. WF $< : defn
```

Since the condition 1 is true, because $(<)$ is well founded, only the condition 0 has to be proved. The condition 0 is similar to the generated PO by VDMTools. The condition is just missing the measure function.

To add this conditions to the goal stack (the goals to be proven), the command *Defn.tgoal* has to be used:

```
- Defn.tgoal fac_defn
> ...
  Proof manager status: 1 proof.
  1. Incomplete:
      Initial goal:
      ?R. WF R /\ !x. ~(x = 0) ==> R (x - 1) x
```

So now using the tactic to prove well formedness of a relation we can start the proof:

```
- e(WF_REL_TAC `measure id`);
OK..
1 subgoal:
> val it =
    !x. ~(x = 0) ==> id (x - 1) < id x
```

Now it is exactly identical. What was done in the VDMTools PO generation was basically simplifying these steps.

After showing the differences between both approaches, this chapter will only concentrate on the first one. The ones generated by VDMTools and translated used *VDM to HOL* translator because that was the focus of the thesis. Both approaches can be undertaken in a similar way as they are practically the same, so there is no point in explaining both.

6.2 Simple Recursion

Simple recursion functions (as mentioned in section 3.6.1) are normally simple to prove valid or invalid. This is the class that HOL easily proves automatically too. The proof normally consists on just making rewrites of the definition and apply arithmetic rules.

Going back to the factorial example that has the following PO that can be introduced into the goal stack like shown below. Since this is the first example and it is still relatively simple, the definitions presented were obtained using the *VDM to HOL* translator directly.

```
- Define `id (id_parameter_1:num) = (let x = id_parameter_1 in x)`;
Definition has been stored under "id_def".
> val id_def =
    |- !id_parameter_1. id id_parameter_1
      = (let x = id_parameter_1 in x) : thm

- BasicProvers.export_rewrites(["id_def"]);
> val it = () : unit

- g `(!  uni_0_var_1.
((((inv_num uni_0_var_1)  /\ (?  n.(n  = uni_0_var_1 ))  )  /\ T )  ==>
 (let n = uni_0_var_1 in
 ((~ (n  = 0 )) ==> ((id n)  > (id (n  - 1 )) ) )) ))`;
> ...
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !uni_0_var_1.
           (inv_num uni_0_var_1 /\ ?n. n = uni_0_var_1) /\ T ==>
           (let n = uni_0_var_1 in ~(n = 0) ==> id n > id (n - 1))
```

This sort of PO can be proven by just applying rewrite tactics.

```
- e (RW_TAC arith_ss [boolTheory.LET_DEF,(fetch "-" "id_def");
Ok..
```

```
...
   Initial goal proved.
   |- !uni_0_var_1.
        (inv_num uni_0_var_1 /\ ?n. n = uni_0_var_1) /\ T ==>
        (let n = uni_0_var_1 in ~(n = 0) ==> id n > id (n - 1))
```

6.3 Nested Recursion

As mentioned in section 3.6.2, the nested recursion can be divided into two different kinds. But this distinction is only concerned about how to generate POs.

6.3.1 "Simple" Nested Recursion

As mentioned in section 3.6.2, the PO generated for this type of function involves proving that the argument of the call after being transformed by other function is still smaller then the input. The example of will show *log* what has to be made. The function *half* as an auxiliary to *log* and it is of the class of simple recursion thus easy to prove that terminates. What is more interesting it is the way to prove that *log* terminates. The function *log* is defined as following:

```
log : nat -> nat
log(n) ==
  if (n = 0) then 0
  else if (n = 1) then 0
  else 1 + log(half(x-2) + 1)
measure id;

half : nat -> nat
half(n) ==
  if (n = 0) then 0
  else if (n = 1) then 0
  else 1 + half(n-2)
measure id;
```

The PO for *log* is:

```
forall n:nat & not(n=0) and not(n=1) => id(n) > id(1 + half(n-2))
```

Which translated to HOL is the following:

```
!n:num . ~(n=0) /\  ~(n=1) ==> id(n) > id(1 + half(n-2))
```

To prove it in HOL, firstly the termination of *half* should be proved. Being *half* a simple function the termination proof can be easily archived. Having proved the termination of half, the following lemma, called induction lemma, can be very useful in the termination proof of *log*.

```
!(n:num). id(n) >= id(half(n))
```

This lemma is useful because it facilitates an almost direct proof to *log* termination. Normally, induction lemmas like this can be proven by recursive induction. In HOL this can be made using the tactic *recInduct* over the induction theorem of half. To get the induction theorem of *half*, the function must be defined in HOL.

```
- Define `half (half_parameter_1:num)  =
  (let n = half_parameter_1 in
    (if (n  = 0 ) then 0
     else (if (n  = 1 ) then 0 else (1  + (half (n  - 2 )) ))
     )
   )`;
> ...
```

This can only be made this way if the termination of the function in question can be automatically proved in HOL. Otherwise it will involve defining the function using the *Hol_defn* command and the subsequent manual proof by the user of the termination clause. It is only possible to extract the induction theorem after the termination is proved.

```
- g `!(n:num). id(n) >= id(half(n))`;
> ...
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !n. id n >= id (half n)
```

Using recursive induction and rewrites it is normally possible to prove this kind of property. It takes a lot of tactics but they are all basically rewrites:

```
- e ( REWRITE_TAC [(fetch "-" "id_def")] THEN
  FULL_SIMP_TAC arith_ss [boolTheory.LET_THM] THEN
  recInduct (fetch "-" "half_ind") THEN
  REPEAT STRIP_TAC THEN
  RW_TAC arith_ss [Once (fetch "-" "half_def")] THEN
  FULL_SIMP_TAC arith_ss []);
> val it =
    Initial goal proved.
    |- !n. id n >= id (half n)

- val half_leq = top_thm();
> val half_leq = |- !n. id n >= id (half n) : thm
```

The first thing to do is rewrite the measure function and simplify the 'let' expressions. Afterwards the induction lemma should be applied. And with a couple of

rewrites more and the goal is proved. In the end the theorem is given a name (in this case *half_leq* and removed from the stack using the command *'top_thm'*. This property about *half* will enable the following reasoning. the estimation of '$\mathrm{half}(x)$' by x. So the following hypothesis can be added to the assumptions of the PO of *log*.

$$
\begin{array}{ll}
 & \forall x : \mathrm{nat}.\ x \neq 0 \wedge x \neq 1 \implies \mathrm{id}(x) > \mathrm{id}(1 + \mathrm{half}(x-2)) \\
\implies & \{ \text{ because of Induction Lemma } \mathrm{id}(x) > \mathrm{id}(\mathrm{half}(x)) \} \\
 & \forall x : \mathrm{nat}.\ x \neq 0 \wedge x \neq 1 \implies \mathrm{id}(x) > \mathrm{id}(1 + (x-2)) \\
\implies & \{ \text{ arithmetic rewrites and } \mathrm{id} \} \\
 & \forall x : nat.\ x \neq 0 \wedge x \neq 1 \implies x > x - 1 \\
\implies & \{ \text{ trivial } \} \\
 & \text{truth}
\end{array}
$$

In HOL this technique can be used also, it requires the *ASSUME_TAC*. Going back to the *log* PO:

```
forall n:nat & not(n=0) and not(n=1) => id(n) > id(1 + half(n-2))
```

Which translated to HOL is:

```
- g `(!  uni_0_var_1.((((inv_num uni_0_var_1)
  /\ (?  n.(n  = uni_0_var_1 )) )  /\ T )  ==>
  (let n = uni_0_var_1 in (((~ (n  = 0 ))  /\ (~ (n  = 1 )) )
  ==> ((id n)  > (id (1  + (half (n  - 2 )) )) ) )) ))`;
> ...
```

The tactics needed to prove this PO are:

```
- e ( REPEAT STRIP_TAC THEN
  VDM_REWRITE_TAC () THEN
  ASSUME_TAC (Q.SPEC `n-2` half_leq) THEN
  FULL_SIMP_TAC arith_ss [(fetch "-" "id_def"),boolTheory.LET_THM] );
>  ...
  Initial goal proved.
```

This is what it takes to prove that a PO is valid for this kind of function. The additional tactic used here was the one mentioned before, *ASSUME_TAC* allows that a assumption is made based on a proven theorem. The theorem has to be specialized (using *Q.SPEC*) to the argument that matters, in this case '*n-2*'. With this assumption it is easy to prove the first goal by estimation of the input of half.

6.3.2 Process of proof with Induction Lemma

To prove the termination of a nested recursive functions of this class, the following properties must be verified:

- the termination of the nested function must be proved first.
- the position where the nested function occurs must be monotonic with the measure. This means that increasing the input value of the nested function must also increase the measure.
- the property that the return of the nested function is smaller or equal to the input is called *Induction Lemma* and it needs to be proved so that the estimation of the nested function can be made.

6.3.3 Cases where the Induction Lemma is not useful

The following example illustrates a case where an Induction Lemma is useless.

```
pred : nat -> nat
pred(n) ==
  if n = 1 then 0
  else 1 + pred(n-1)
pre n > 0;

foo : nat -> nat
foo(n) ==
  if n = 0 then 0
  else foo(pred(n))
measure id;
```

The termination of *pred* is simple to prove. The problem appears when it is attempted to prove the termination of *foo* using the estimation technique. The PO for the termination of *foo* is the following in VDM:

```
forall n:nat & not(n=0) => id(n) > id(pred(n))
```

The induction lemma for this PO is the following in HOL:

```
!(n:num). id(n) >= id(pred(n))
```

This is, the induction lemma is not strong enough to help to prove the PO. In fact, it is even weaker then the PO. So in this cases the approach to follow is attempt to prove the PO directly. The PO for *foo* would be:

```
!(n:num). pre_pred(n) /\ ~(n=1) ==> id(n) > id(pred(n))
```

This means that *pred* needs also to be defined in HOL.

```
- val pre_pred_def =
```

```
  Define `pre_pred n = n > 0`;
> ...
- TotalDefn.default_termination_simps
  := (fetch "-" "pre_pred_def") :: !TotalDefn.default_termination_simps;
> ...
- val pred_defn =
  Define `pred (n:num)  =
    if pre_pred n then
      ((if (n  = 1 ) then 0 else (1  + (pred (n  - 1 )) )))
    else ARB`;
>
```

Since these are the simplified version of the definitions of the functions, the proof of the ones that come directly from the *VDM to HOL* translator need to use the tactic that simplifies the *let* statements. This is the only difference between them. With that induction theorem, the PO can be tackled. This PO also demonstrate why it is difficult to automate these proofs. Applying recursive induction is not enough because after some rewrites the result is this one:

```
- e ( REWRITE_TAC [pre_pred_def,id_def] THEN
  recInduct pred_ind THEN
  REPEAT STRIP_TAC THEN
  FULL_SIMP_TAC arith_ss [pre_pred_def] THEN
  RW_TAC arith_ss [Once pred_def] );
> ...
 n > pred (n - 1) + 1
    ------------------------------------
      0.  ~(n - 1 = 1) ==> n > pred (n - 1) + 1
      1.  n > 0
      2.  ~(n = 1)
      3.  pre_pred n
     : goalstack
```

What can be seen here is that this subgoal is almost proved. Condition (0.) implies what it needs to be proven. The assumption says if $n \neq 2$ then what is to be proved is valid. Yet in case $n = 2$ then the property is trivially true too, but HOL cannot deduct that automatically. This subgoal needs to be split into two cases, the one that $n = 2$ and the one that $\neg(n - 1 = 1)$. This can be made with the *DECIDE_TAC*. Using it and doing some simplification after the result is:

```
- e ( `(n=2) \/ ~(n - 1 = 1)` by DECIDE_TAC THEN
 FULL_SIMP_TAC arith_ss []);
OK..
1 subgoal:
> val it =
    2 > pred 1 + 1
    ------------------------------------
```

```
     0.  T
     1.  T
     2.  T
     3.  pre_pred 2
     4.  n = 2
    : goalstack
```

Which is provable by just rewriting the *pred* definition.

6.3.4 Nested recursion over itself

The second type of nested recursion is when the function calls itself several times in a nested recursion. This type can be divided furthermore as functions that do not need their own semantic considered for termination have a different approach than the ones whose semantics need to be considered .

Proof Obligations that do not involve semantics of nested function

Some nested functions do not need to the nested definition to be explored. The well known Ackerman function can elucidate this point. This is also a typical function that termination can only be proved by using a lexicographic order.

```
id2 : nat * nat -> nat * nat
id2(m,n) == mk_(m,n);

ack : nat * nat -> nat
ack(m,n) ==
  if m = 0 then n + 1
  else if n = 0 then ack(m-1,1)
  else ack(m-1,ack(m,n-1))
measure id2;
```

For this function, three POs are generated, corresponding to the three recursive calls:

```
PO1: (forall m : nat, n : nat &
  not (m = 0) => n = 0 =>
  id2(m, n) (LEX2 >) id2(m - 1, 1))

PO2: (forall m : nat, n : nat &
  not (m = 0) => not (n = 0) =>
  id2(m, n) (LEX2 >) id2(m, n - 1))

PO3: (forall m : nat, n : nat &
  not (m = 0) => not (n = 0) =>
  id2(m, n) (LEX2 >) id2(m - 1, ack(m, n - 1)))
```

Currently the OML parser and *VDM to HOL* translator cannot handle the *LEXn* syntax. So this POs have to be translated manually to HOL. The most interesting

one is the last, so only this one will be shown.

```
- open pairTheory (* contains LEX definition *)
> ...
- Define `id2 x y = (x,y)`;
> ...
- g `!(m:num) (n:num). ~(m=0) ==> (n=0)
  ==> ($> LEX $>) (id2 m n) (id2 (m - 1) (ack(m,n - 1)))`;
> ...
- e (RW_TAC arith_ss [(fetch "-" id2_def"),LEX_DEF]);
> ...
  Initial goal proved.
  ...
```

What is happening here is that using a lexicographic order (see section 1.6.1), the left part of the pair is inspected first, if it is enough to prove the PO valid, then there is no need to consider the right part. Thus there is no need to investigate further the nested recursion. Although the motive that makes the lexicographic order indispensable is the second PO in which the first argument is not decreasing but the second one is.

Proofs that involve semantics of the nested function

This is the kind of PO that contains a reference to the function that is being proved and cannot be solved by the method mentioned above. This kind of function termination proof is extremely difficult to automate since they normally need a provided property that varies from function to function. Take for example the function *nest*:

```
nest : nat -> nat
nest(n) ==
  if n = 0 then 0
  else nest(nest(0))
measure id;
```

Which originates the following POs:

```
PO1:
(forall x : nat & not (x = 0) =>  id(x) > id(x - 1))

PO2:
(forall x : nat & not (x = 0) => id(x) > id(nest(x - 1)))
```

Trying to import the second PO to HOL is somehow pointless. The problems is that in HOL to be able to manipulate a function definition, that definition should be proved terminating. So trying to prove this PO in HOL implies that *nest* is already defined and proved terminating. So unless the PO is used for pen and

paper proofs, it does not make sense to translate it to HOL. What can be done instead is translate the function definition and then prove the termination using HOL mechanisms. Further information on the method that was used can be obtained in [47].

```
- val nest_defn =
  Hol_defn "nest" `nest (n:num) =
    (if (n = 0 ) then 0 else (nest (nest (n  - 1))))`;
> ...
 Termination conditions :
      0. !n. ~(n = 0) ==> R (n - 1) n
      1. !n. ~(n = 0) ==> R (nest_aux R (n - 1)) n
      2. WF R : defn
```

HOL creates *nest_aux* in order to facilitate this process, it represents the inner recursion of the *nest* function. The tactic here is to prove that *nest_aux* is terminating. First the ($<$) relation must be instantiated in the termination conditions using the command *set_reln*.

```
- val nest_aux_defn =
  Defn.set_reln (valOf (Defn.aux_defn nest_defn)) ``(<)``;
> ...
 Termination conditions :
      0. WF $<
      1. ~(n = 0) ==> n - 1 < n
      2. ~(n = 0) ==> nest_aux $< (n - 1) < n : defn
```

The restrains from the induction theorem of *nest_aux* can now be removed by using the command *prove_tcs*

```
- val nest_aux_defn' =
  Defn.prove_tcs nest_aux_defn (SIMP_TAC arith_ss [prim_recTheory.WF_LESS]);
> ...
  Induction :
    |- !P.
         (!n.
            (~(n = 0) ==>
             nest_aux $< (n - 1) < n ==>
             P (nest_aux $< (n - 1))) /\ (~(n = 0) ==> P (n - 1)) ==>
            P n) ==>
         !v. P v
  ...
```

With the induction theorem, the property that enables the proof of the outer nest can be now proved. The property is that basically the inner nest computes the function constant 0. The first two commands extract the induction theorem and definition from *Defn* structure like *nest_aux_defn'*.

```
- val nest_aux_ind = valOf (Defn.ind_of nest_aux_defn');
> ...
- val [E] = Defn.eqns_of nest_aux_defn';
> ...
- val nest_aux_thm = Q.prove
   (`!n. nest_aux (<) n = 0`,
    recInduct nest_aux_ind THEN
    REPEAT STRIP_TAC THEN RW_TAC arith_ss [DISCH_ALL E]);
> val nest_aux_thm = |- !n. nest_aux $< n = 0 : thm
```

With this theorem in hand is then easy to prove the termination of *nest*:

```
- val (nest_def,nest_ind) =
     Defn.tprove
     (defn, WF_REL_TAC `$<` THEN RW_TAC arith_ss [nest_aux_thm]);
> val nest_def = |- nest n = (if n = 0 then 0 else nest (nest (n - 1)))
   : thm
  val nest_ind =
    |- !P.
         (!n.
            (~(n = 0) ==> P (nest (n - 1))) /\ (~(n = 0) ==> P (n - 1)) ==>
            P n) ==> !v. P v : thm
```

Finally *nest*'s termination is proved and induction theorem and definition are available for use.

As it was seen here, this kind of proof for nested functions is complex and it changes from function to function. For a proof like this, all the details and properties have to be custom crafted. A good thing is that in computation not many nested recursive functions are used, mainly because they are subject to failure due to filling the memory stack and are inefficient [8].

6.4 Mutual Recursion

Mutual recursion does not raise any problem different from the ones explained above. The fact is that generated POs for mutual recursion fall onto the presented categories as they do not add anything distinct. The only variance is that to prove the termination of mutual recursive functions, the POs for both of them must be proved.

The following *even/odd* example is a typical mutual recursion:

```
m_even : nat -> nat
m_even(x) == 2 * x + 1;

even : nat -> bool
even(n) ==
```

```
  if n = 0 then true
  else odd(n)
measure m_even;

m_odd : nat -> nat
m_odd(x) == 2 * x;

odd : nat -> bool
odd(n) ==
  if n = 0 then false
  else even(n-1)
measure m_odd;
```

The generated POs are:

```
PO1: (forall n : nat & not (n = 0) => m_odd(n) > m_even(n - 1))

PO2: (forall n : nat & not (n = 0) => m_even(n) > m_odd(n))
```

These POs are simple, they do not involve references to the functions in question, so they can be solved by just applying rewrite tactics. If POs take this form, this is, without referring to the actual functions then they can be translated with the *VDM to HOL* translator. Otherwise if any of the mutual recursive functions is referred, then the translator can be used to translate the functions definitions, but both declarations of the functions have to be joined manually. The result of the translator result is correct, though in HOL, mutual recursive functions have to be declared at the same time and not in different declaration like the translators outcome. Again, these POs are redundant if the functions have to be defined in HOL, since they need to be prove terminating even before the definitions being used.

Chapter 7

Concluding Remarks and Related Work

7.1 Assessment of the Project Results

The aims of this project were presented in section 1.7 and they were accomplished. The specification of the POG of VDMTools was successfully updated to generate termination proof obligations and are already implemented and being used by users all over the world. So a missing set of proof obligations that were missing in the VDMTools proof obligation generator was analyzed and specified. The proof obligations for termination of recursive functions is another tool at the specifier hands. Proving these will enable the specifier to have more confidence in the design. In critical applications, the certainty that a program is not going to enter in an infinite loop is a definitely a major advance.

In the study of HOL limits to prove these kind of proof obligations some conclusions were reached. First that it is possible to automate some kinds of simple proofs. There are also cases where a clearly terminating function is difficult to tackle automatically. The proof of termination of nested functions is very difficult to implement as automatic in HOL, if not impossible currently. This is because these functions normally require some auxiliary properties to be proven, that cannot be automatically generated. However, for all the proof obligations cases, guidelines were provided to aid the users use of this tool.

A framework for termination proofs was laid. This is an important step for users that want to archive complete formality. In addition, these advances in proof automatization enable a specifier to be more formal with less hassle, i.e. some termination proofs can be made automatically and in this way confidence is gained by just pressing a ”proof button”. Furthermore a user can follow the the guidelines presented in this work and try to prove that the specification has

no inconsistencies for the ones that cannot be proven automatically.

7.2 Future Work

Completion of code: For a few exceptional cases, for example, recursive functions are not detected if they are defined using a super class function.

Automatic measure generation: From a user point of view, for more complex recursion it can be discouraging to have to learn how to construct measures that enable the validity of the proof obligation. So techniques for generate measure automatically could be studied in order to ease up this task.

Automatic proofs: The ultimate goal is of course, the automatic proof of all proof obligations. With improvement in tactic automation and in theorem provers this might become a reality in the future. Making proofs invisible to the user should be the aim as some termination proofs are intrinsically difficult and this is a big drawback since it requires potential users to learn termination notions and also as proofs can be arbitrarily hard, it demands that users also learn about theorem provers [40].

POG implementation: In respect to the implementation performance it is fair to say it could optimized further. This matter will require analysis of the bottlenecks at specification level. In addition one can find faster algorithms to perform the same tasks or even a different approach to how functions should be checked for recursion should be studied.

Eclipse integration: Overture Tools are developed as Eclipse plug-ins. An integration of the work in this project and Eclipse could lead to a better user experience proving proof obligations.

Non-functional subset: The approach shown in this thesis is valid for the VDM functional subset. In addition one could investigate would to generate termination proof obligations for `while` or `for` loops.

VDM to HOL translator: As mention in chapter 5, some improvements could be made to the translator if the recursion map generated in the VDM type checker could somehow be exported. With this information, various improvements could be made in the translation.

7.3 Related Work

Some similar work has been done before, but for VDM this is the first attempt to generate termination proof obligations.

VDM integrity checker: It generates proof obligations for parts of the model that cause inconsistency although termination of recursive functions is not contemplated [28, 5, 3].

VDM to HOL: the translator from VDM specifications into HOL and tactics to prove generated proof obligations was defined in [53]. It can be used to prove static inconsistencies in VDM models. However termination inconsistencies are not considered in this work.

PROSPER: was a Proof Engine developed in HOL98 which aimed to support formal proofs for industry-standard languages, like VDM and VHDL, by making proofs invisible to the end user [15, 14]. A translator from VDM to HOL was defined in [1] and also tactics to prove the generated proof obligations [2].

PVS: PVS (Prototype Verification System) [44] is an environment for specification and verification consisting of a specification language, a parser, a type checker and an interactive theorem prover. The PVS type checker generates TCCs (Type Correctness Conditions) that contemplate termination of recursive functions. However PVS allows less expressiveness because the modeling of mutual recursive functions is not allowed [43].

TFL: Terminating Functional Programs [46], is an environment for defining and reasoning about termination programs written in a purely functional manner. It extracts termination conditions from a program and tries to prove them. This system was integrated into Isabelle and HOL.

Bibliography

[1] S. Agerholm and K. Sunesen. Formalizing a subset of vdm-sl in hol. Technical report, IFAD, April 1999. [cited at p. 67]

[2] S. Agerholm and K. Sunesen. Reasoning about VDM-SL Proof Obligations in HOL. Technical report, IFAD, 1999. [cited at p. 67]

[3] S. Agerholm and K. Sunesen. Vdm-sl toolbox extensions. Technical report, IFAD, April 1999. [cited at p. 67]

[4] Sten Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, Boppard, Germany, October 1998. Springer-Verlag. [cited at p. 4]

[5] Bernhard Aichernig. A Proof Obligation Generator for the IFAD VDM-SL Toolbox. Master's thesis, Technical University Graz, Austria, March 1997. [cited at p. 25, 67]

[6] Tiago M. L. Alves. Sex up overture. In *Third Overture Workshop at Newcastle University*, November 2006. [cited at p. 6]

[7] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X. [cited at p. 4]

[8] Scott Chasalow and Richard Brand. Generation of simplex lattice points. *U.C. Berkeley Division of Biostatistics Working Paper Series*, August 1992. [cited at p. 63]

[9] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementations. *Science of Computer Programming*, 9(1), 1987. [cited at p. 11]

[10] Krzysztof Ciesielski. *Set Theory for the Working Mathematician*. Cambridge University Press, 1997. [cited at p. 11]

[11] Peter Linsay Cliff Jones, Kevin Jones and Richard Moore, editors. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X. [cited at p. 9]

[12] Flemming M. Damm and Bo Stig Hansen. Generation of Proof Obligations for Type Consistency. Technical Report 1993-123, Department of Computer Science, Technical University of Denmark, December 1993. [cited at p. 25]

[13] D. de Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19(20):199–260, 1994. [cited at p. 11]

[14] Louise Dennis and Tom Melham. Prosper technology roadmap. Technical report, University of Glasgow, 1999. [cited at p. 67]

[15] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER Toolkit. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. Springer-Verlag, Lecture Notes in Computer Science volume 1785. [cited at p. 67]

[16] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0. [cited at p. 22]

[17] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005. [cited at p. 4, 22]

[18] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: advances in support for formal modeling in VDM. Submitted for publication 2007. [cited at p. 4, 5]

[19] Hans Bruun Flemming Damm and Bo Stig Hansen. On Type Checking in VDM and Related Consistency Issues. In *VDM '91: Formal Software Development Methods*, pages 45–62. VDM Europe, Springer-Verlag, October 1991. [cited at p. 25]

[20] Jürgen Giesl. Automated termination proofs with measure functions. In *Proceedings of the 19th Annual German Conference on Artificial Intelligence*. Springer-Verlag, 1995. [cited at p. 11]

[21] Jürgen Giesl. Termination analysis for functional programs using term orderings. *Lecture Notes in Computer Science*, 983, 1995. [cited at p. 10, 11, 12]

[22] Jürgen Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19:10–29, 1997. [cited at p. 10]

[23] Mike Gordon. From lcf to hol: a short history. University of Cambridge Computer Laboratory, October 1996. [cited at p. 15, 16]

[24] Cambridge HOL group. *The HOL System: Reference [For HOL Kananaskis-4]*. University of Cambridge. http://hol.sourceforge.net/. [cited at p. 15, 18, 20]

[25] Cambridge HOL group. *The HOL System: Description [For HOL Kananaskis-4]*. University of Cambridge, January 2007. http://hol.sourceforge.net/. [cited at p. 16]

[26] Cambridge HOL group. *The HOL System: Tutorial [For HOL Kananaskis-4]*. University of Cambridge, January 2007. http://hol.sourceforge.net/. [cited at p. 17]

[27] L. Hatton. *Safer C: Developing Software for High-integrity and Safety-critical Systems*. McGraw-Hill International, 1994. [cited at p. 6]

[28] IFAD. *VDMTools: The VDM-SL Integrity Examiner*. IFAD, 2000. [cited at p. 67]

[29] Information Technology Programming Languages – VDM-SL. Technical report, First Committee Draft Standard: CD 13817-1, November 1993. ISO/IEC JTC1/SC22/WG19 N-20. [cited at p. 4]

[30] Cliff B. Jones. VDM Proof Obligations and their Justification. In Airchinnigh Bjørner, Jones and Neuhold, editors, *VDM '87 VDM – A Formal Method at Work*, pages 260–286. VDM-Europe, Springer-Verlag LNCS 252, 1987. [cited at p. 3, 4]

[31] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7. [cited at p. 4]

[32] Fairouz Kamareddine and Francois Monin. On formalised proofs of termination of recursive functions. *Lecture Notes in Computer Science*, 1702:29–46, 1999. [cited at p. 10, 11]

[33] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996. [cited at p. 4]

[34] Peter Gorm Larsen and Wiesław Pawłowski. The Formal Semantics of ISO VDM-SL. *"Computer Standards and Interfaces"*, 17(5–6):585–602, September 1995. [cited at p. 8]

[35] Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford University, Stanford, CA, USA, 1972. [cited at p. 15]

[36] Moscow Meta Language. http://www.itu.dk/ sestoft/mosml.html, 2007. [cited at p. 16]

[37] Magnus O. Myreen. *Guide to HOL4 interaction and basic proofs*. http://www.cl.cam.ac.uk/˜mom22/. [cited at p. 17]

[38] National Institute of Standards and Technology (NIST). NIST Web site. http://www.nist.gov, 2007. [cited at p. 9]

[39] Overture-Core-Team. Overture Web site. http://www.overturetool.org, 2007. [cited at p. 5]

[40] Scott Owens and Konrad Slind. Adapting functional programs to higher-order logic. *To appear in Higher-order and Symbolic Computation*, 2007. [cited at p. 66]

[41] Nico Plat. The Industrial use of VDM++. In *IEE Colloquium on Industrial Use of Formal Methods*. IEE, May 1997. [cited at p. 4]

[42] Robin Milner Robert Harper and Mads Tofte. The Definition of Standard ML. Technical report, Department of Computer Science, University of Edingburgh, The King's Building, 1989. [cited at p. 16]

[43] J. M. Rushby S. Owre, N. Shankar and D. W. J. Stringer-Calvert. *PVS Language Reference - Version 2.4*, 2001. [cited at p. 67]

[44] J. M. Rushby S. Owre, N. Shankar and D. W. J. Stringer-Calvert. *PVS System Guide - Version 2.4*, 2001. [cited at p. 67]

[45] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986. [cited at p. 6]

[46] Konrad Slind. Function definition in higher-order logic. [cited at p. 67]

[47] Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Technische Universität München, 1999. [cited at p. 62]

[48] ICL Secure Systems. Proofpower. http://www.lemma-one.com/ProofPower/index/index.html. [cited at p. 16]

[49] P. van der Spek. The overture project: Designing an open source tool set. Master's thesis, Delf University of Technology, August 2004. [cited at p. 5]

[50] P. van der Spek. The overture project: Towards an open source toolset. Technical report, Delf University of Technology, January 2004. [cited at p. 5]

[51] Marcel Verhoef and Peter Gorm Larsen. Enhancing VDM++ for Modeling Distributed Embedded Real-time Systems. Technical Report (to appear), Radboud University Nijmegen, March 2006. A preliminary version of this report is available on-line at *http://www.cs.ru.nl/∼marcelv/vdm/*. [cited at p. 4]

[52] Marcel Verhoef and Peter Gorm Larsen. Interpreting Distributed System Architectures Using VDM++ – A Case Study. In Brian Sauser and Gerrit Muller, editors, *5th Annual Conference on Systems Engineering Research*, March 2007. Available at http://www.stevens.edu/engineering/cser/. [cited at p. 6]

[53] Sander Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department, August 2007. [cited at p. 6, 13, 47, 67]

Appendices

Appendix A

"Rec" Module

This is part of my work in VDM, the **REC** module defines important functions that were used for the generation of the terminations proof obligations.

```
\section{Recursive definitions}
This module contains everything that is needed for the static
checker to set the framework for recursive PO. It contains
functions to build the recursion map, that will be used for
generate warnings or errors and also will be later used by
the POG to generate the POs.

\begin{vdm_al}
module REC

imports
  from PAT all,
  from ERR all,
  from AS all,
  from CHECK all,
  from CI all,
  from REP all,
  from DEF all,
  from ENV all,
  from VCM all


exports
  all

definitions


functions
```

```
\end{vdm_al}
This functions do the transitive closure process while
mantaining the trace of the recursive functions called.
First the sets inside the \emph{FunctionInfo} are
transformed in lists and then the transitive closure
process begins.

\begin{vdm_al}
  -- MAIN function invokes all the others
  main_RecMap : map AS`Name to ENV`FunctionInfo
    -> map AS`Name to ENV`FunctionInfo
  main_RecMap(fnm) ==
   disting_recMap(trans_RecMap(recMapSet2Seq(fnm)),fnm);

  -- distinguish between direct recursion or mutual recursive definitions
  -- in the end if the key is equal to the last element of the lists the head
  -- of the trace should be returned
  -- the head is either the key (direct recursion) or other function
  -- (mutual recursion)
  disting_recMap : map AS`Name to set of seq of AS`Name
    * map AS`Name to ENV`FunctionInfo
      -> map AS`Name to ENV`FunctionInfo
  disting_recMap(fnm,recmap) ==
    { x |-> mk_ENV`FunctionInfo({ hd y | y in set fnm(x) & y(len y) = x },
     recmap(x).meas ,recmap(x).printPO) | x in set dom fnm }
  pre forall y in set rng fnm & y inter {[]} = {};

  -- transforms the recursion map {fname |-> {calledfns}} to
  -- {fname |-> {[fn1],..,[fnN]}}
  recMapSet2Seq : map AS`Name to ENV`FunctionInfo
    -> map AS`Name to set of seq of AS`Name
  recMapSet2Seq(fnm) ==
    { x |-> { [y] | y in set fnm(x).rec } | x in set dom fnm};

\end{vdm_al}
Make the transitive closure of a map but keeping the traces so the
recursive functions can be identified.
\begin{vdm_al}
  trans_RecMap : map AS`Name to set of seq of AS`Name
    -> map AS`Name to set of seq of AS`Name
  trans_RecMap(recm) ==
    let recm' = compose_RecMap(recm,recm)
    in if recm = recm' then recm
       else trans_RecMap(recm');

\end{vdm_al}
Auxiliary functions for map composition.
\begin{vdm_al}
  compose_RecMap : (map AS`Name to set of seq of AS`Name)
    * (map AS`Name to set of seq of AS`Name)
     -> map AS`Name to set of seq of AS`Name
```

```
  compose_RecMap(m,n) ==
    { x |-> compose_auxRecMap(m(x),n) | x in set dom m};

  compose_auxRecMap : set of seq of AS`Name
    * (map AS`Name to set of seq of AS`Name)
      -> set of seq of AS`Name
  compose_auxRecMap(trac,fnm) ==
    dunion { compose_aux2RecMap(x,fnm) | x in set trac };


  compose_aux2RecMap : seq of AS`Name
    * (map AS`Name to set of seq of AS`Name)
    -> set of seq of AS`Name
  compose_aux2RecMap(trace,fnm) ==
    let key = trace(len trace)
    in if key in set dom fnm
      then { if {hd x} inter elems trace = {} then trace^[hd x]
                 else trace | x in set fnm(key)  }
      else { trace };


\end{vdm_al}
Finds all the applications on a function body. It really finds
all identifiers but after they are intersected with function
names in order to keep only the functions.
\begin{vdm_al}
getFuncAppFnDef : AS`FnDef * set of AS`Name -> set of AS`Name
getFuncAppFnDef(fnd,sn) ==
  cases fnd:
  mk_AS`ExplFnDef(-,-,-,-,-,-,-,-,-,-) -> getFuncAppExplFnDef(fnd,sn),
  others -> {}
  end;

getFuncAppExplFnDef : AS`ExplFnDef * set of AS`Name
  -> set of AS`Name
getFuncAppExplFnDef(fn,sn) ==
  getFuncAppFnBody(fn.body,sn);

getFuncAppFnBody : AS`FnBody * set of AS`Name
  -> set of AS`Name
getFuncAppFnBody(mk_AS`FnBody(body,-),sn) ==
  cases body:
  <NOTYETSPEC> -> {},
#ifdef VDMPP
  <SUBRESP>    -> {},
#endif VDMPP
  others       -> getFuncAppExpr(body,sn)
  end;

getFuncAppExpr : AS`Expr * set of AS`Name -> set of AS`Name
getFuncAppExpr(expr,sn) ==
```

```
cases expr :
  mk_AS`DefExpr(-,-,-)
    -> getFuncAppDefExpr(expr,sn),
  mk_AS`LetExpr(-,-,-)
    -> getFuncAppLetExpr(expr,sn),
  mk_AS`LetBeSTExpr(-,-,-,-)
    -> getFuncAppLetBeSTExpr(expr,sn),
  mk_AS`IfExpr(-,-,-,-,-)
    -> getFuncAppIfExpr(expr,sn),
  mk_AS`CasesExpr(-,-,-,-)
    -> getFuncAppCasesExpr(expr,sn),
  mk_AS`PrefixExpr (-,-,-)
    -> getFuncAppPrefixExpr(expr,sn),
  mk_AS`MapInverseExpr (e,-)
    -> getFuncAppExpr(e,sn),
  mk_AS`BinaryExpr(-,-,-,-)
    -> getFuncAppBinaryExpr(expr,sn),
  mk_AS`AllOrExistsExpr(-,bind,e,-)
    -> getFuncAppExpr(e,sn) union
       dunion {getFuncAppMultBind(b,sn) | b in set elems bind},
  mk_AS`ExistsUniqueExpr(bind,e,-)
    -> getFuncAppExpr(e,sn) union getFuncAppBind(bind,sn),
  mk_AS`SetEnumerationExpr(exprs,-)
    -> dunion {getFuncAppExpr(e,sn) | e in set elems exprs},
  mk_AS`SetComprehensionExpr(e1,bind,e2,-)
    -> getFuncAppExpr(e1,sn) union (if e2 = nil
                                    then {}
                                    else getFuncAppExpr(e2,sn)) union
       dunion {getFuncAppMultBind(b,sn) | b in set elems bind},
  mk_AS`SetRangeExpr(l,r,-)
    -> getFuncAppExpr(l,sn) union getFuncAppExpr(r,sn),
  mk_AS`SeqEnumerationExpr(seqe,-)
    -> dunion {getFuncAppExpr(x,sn) | x in set elems seqe},
  mk_AS`SeqComprehensionExpr(e1,bind,e2,-)
    -> getFuncAppExpr(e1,sn) union (if e2 = nil
                                    then {}
                                    else getFuncAppExpr(e2,sn)) union
       getFuncAppBind(bind,sn),
  mk_AS`SubSequenceExpr(e1,e2,e3,-)
    -> getFuncAppExpr(e1,sn) union getFuncAppExpr(e2,sn)
      union getFuncAppExpr(e3,sn),
  mk_AS`SeqModifyMapOverrideExpr(e1,e2,-)
    -> getFuncAppExpr(e1,sn) union getFuncAppExpr(e2,sn),
  mk_AS`MapEnumerationExpr(els,-)
    -> dunion {getFuncAppMaplet(e,sn) | e in set elems els},
  mk_AS`MapComprehensionExpr(-,-,-,-)
    -> getFuncAppMapCampExpr(expr,sn),
  mk_AS`TupleConstructorExpr(-,-)
    -> getFuncAppTupleConstructorExpr(expr,sn),
  mk_AS`TupleSelectExpr(e,-,-)
    -> getFuncAppExpr(e,sn),
```

```
    mk_AS`TypeJudgementExpr(e,-,-)
      -> getFuncAppExpr(e,sn),
    mk_AS`PreConditionApplyExpr (-,seqe,-)
      -> dunion { getFuncAppExpr(e,sn) | e in set elems seqe },
    mk_AS`TokenConstructorExpr(e,-)
      -> getFuncAppExpr(e,sn),
    mk_AS`RecordConstructorExpr (-,fields,-)
      ->  dunion { getFuncAppExpr(e,sn) | e in set elems fields },
    mk_AS`RecordModifierExpr (e,mods,-)
      ->  getFuncAppExpr(e,sn) union
          dunion {getFuncAppRecordModification(m,sn) | m in set elems mods},
    mk_AS`ApplyExpr (-,-,-)
      -> getFuncAppApplyExpr(expr,sn),
    mk_AS`FieldSelectExpr(rec,-,-)
      -> getFuncAppExpr(rec,sn),
    mk_AS`LambdaExpr(tpbind,e,-)
      -> getFuncAppExpr(e,sn) union
         dunion {getFuncAppBind(bind,sn) | bind in set elems tpbind},
    mk_AS`IsExpr (-,e,-)
      -> getFuncAppExpr(e,sn),
    mk_AS`BoolLit(-,-),
    mk_AS`RealLit(-,-),
    mk_AS`CharLit(-,-),
    mk_AS`TextLit(-,-),
    mk_AS`QuoteLit(-,-),
    mk_AS`NilLit(-) -> {},
    mk_AS`Name(-,-) -> {expr} \ sn,
    mk_AS`UndefinedExpr(-) -> {},
    mk_AS`IotaExpr(bind,e,-)
          -> getFuncAppExpr(e,sn) union getFuncAppBind(bind,sn),
    mk_AS`FctTypeInstExpr(-,-,-) -> {},
    mk_AS`BracketedExpr(e,-)
      -> getFuncAppExpr(e,sn),
--    mk_AS`ExplFnDef(-,-,-,-,-,-,-,-,-,-,-)
--      -> getFuncAppFnDef(e,sn),
#ifdef VDMPP
    mk_AS`NewExpr(-,exprs,-)
      -> dunion { getFuncAppExpr(e,sn) | e in set elems exprs },
    mk_AS`ThreadIdExpr(-) -> {},
    mk_AS`SelfExpr(-) -> {},
    mk_AS`SameClassExpr(-,-,-),
    mk_AS`SameBaseClassExpr(-,-,-) -> {},
    mk_AS`IsOfClassExpr(-,-,-),
    mk_AS`IsOfBaseClassExpr(-,-,-),
    mk_AS`ActExpr(-,-),
    mk_AS`FinExpr(-,-),
    mk_AS`ActiveExpr(-,-),
    mk_AS`WaitingExpr(-,-),
    mk_AS`ReqExpr(-,-) -> {},
#ifdef VICE
    mk_AS`CurTimeExpr(-) -> {},
```

```
#endif VICE
#endif VDMPP
        others -> {}
      end;



operations

getFuncAppDefExpr: AS`DefExpr * set of AS`Name ==> set of AS`Name
getFuncAppDefExpr(mk_AS`DefExpr(Def,InExpr,-),sn) ==
  ( dcl ids: set of AS`Name := {},
        remids: set of AS`Name := sn;
    for i = 1 to len Def do
    (
      let expids = getFuncAppExpr(Def(i).#2,remids),
          patids = getFuncAppPatternBind(Def(i).#1)
      in
      (  ids := ids union expids;
         remids := remids union patids
      )
    );
    return getFuncAppExpr(InExpr,remids) union ids
  );

functions

getFuncAppPatternBind : AS`PatternBind -> set of AS`Name
getFuncAppPatternBind(pb) ==
  cases pb:
  mk_AS`SetBind(p,-,-)  -> PAT`ExtractPatternName(p),
  mk_AS`TypeBind(p,-,-) -> PAT`ExtractPatternName(p),
  others                -> PAT`ExtractPatternName(pb)
  end;


getFuncAppLetExpr : AS`LetExpr * set of AS`Name -> set of AS`Name
getFuncAppLetExpr(expr,sn) ==
  let res = getFuncAppLocalDef(expr.localdef,sn)
  in
  res.#1 union getFuncAppExpr(expr.body,res.#2);


operations
getFuncAppLocalDef : seq of AS`LocalDef * set of AS`Name
  ==> set of AS`Name * set of AS`Name
getFuncAppLocalDef(vd,sn) ==
  ( dcl ids: set of AS`Name := {},
        remids: set of AS`Name := sn;
    for i = 1 to len vd do
    (
```

```
      cases vd(i):
      mk_AS`ValueDef(-,-,-,-,-,-) ->
        (
           let expids = getFuncAppExpr(vd(i).val,remids),
               patids = getFuncAppPatternBind(vd(i).pat)
           in
           (  ids := ids union expids;
              remids := remids union patids
           )
        ),
      mk_AS`ExplFnDef(-,-,-,-,-,-,-,-,-,-) ->
        ids := ids union getFuncAppFnDef(vd(i),sn),
      others -> skip
      end;
    );
    return mk_(ids,remids)
  );

functions
getFuncAppApplyExpr : AS`ApplyExpr * set of AS`Name
  -> set of AS`Name
getFuncAppApplyExpr(mk_AS`ApplyExpr(fct,arg,-),sn) ==
  getFuncAppExpr(fct,sn)
  union
  dunion {getFuncAppExpr(x,sn) | x in set elems arg};



getFuncAppIfExpr : AS`IfExpr * set of AS`Name
  -> set of AS`Name
getFuncAppIfExpr(expr,sn) ==
  getFuncAppExpr(expr.test,sn)
  union getFuncAppExpr(expr.cons,sn)
  union
    dunion {getFuncAppElseifExpr(x,sn) | x in set elems expr.elsif}
  union getFuncAppExpr(expr.altn,sn);

getFuncAppElseifExpr : AS`ElseifExpr * set of AS`Name
  -> set of AS`Name
getFuncAppElseifExpr(expr,sn) ==
  getFuncAppExpr(expr.test,sn)
  union
  getFuncAppExpr(expr.cons,sn);

getFuncAppBinaryExpr : AS`BinaryExpr * set of AS`Name -> set of AS`Name
getFuncAppBinaryExpr(mk_AS`BinaryExpr(left,-,right,-),sn) ==
  getFuncAppExpr(left,sn)
  union
  getFuncAppExpr(right,sn);

getFuncAppTupleConstructorExpr : AS`TupleConstructorExpr * set of AS`Name
```

```
  -> set of AS`Name
getFuncAppTupleConstructorExpr(expr,sn) ==
  dunion {getFuncAppExpr(x,sn) | x in set elems expr.fields};

getFuncAppPrefixExpr : AS`PrefixExpr * set of AS`Name -> set of AS`Name
getFuncAppPrefixExpr(expr,sn) ==
  getFuncAppExpr(expr.arg,sn);

getFuncAppCasesExpr : AS`CasesExpr * set of AS`Name -> set of AS`Name
getFuncAppCasesExpr(expr,sn) ==
  getFuncAppExpr(expr.sel,sn)
  union
  dunion { getFuncAppCaseAltn(x,sn) | x in set elems expr.altns }
  union
  (if expr.Others = nil then {} else getFuncAppExpr(expr.Others,sn))
  ;

getFuncAppCaseAltn : AS`CaseAltn * set of AS`Name -> set of AS`Name
getFuncAppCaseAltn(expr,sn) ==
  let a = dunion {getFuncAppPattern(pat,sn) | pat in set elems expr.match},
      b = getFuncAppExpr(expr.body,sn)
  in
    a union b;


getFuncAppLetBeSTExpr : AS`LetBeSTExpr * set of AS`Name -> set of AS`Name
getFuncAppLetBeSTExpr(expr,sn) ==
  getFuncAppBind(expr.lhs,sn)
  union if expr.St = nil then {} else getFuncAppExpr(expr.St,sn)
  union getFuncAppExpr(expr.In,sn);

getFuncAppMaplet : AS`Maplet * set of AS`Name -> set of AS`Name
getFuncAppMaplet(expr,sn) ==
  getFuncAppExpr(expr.mapdom,sn)
  union getFuncAppExpr(expr.maprng,sn);

getFuncAppMapCampExpr : AS`MapComprehensionExpr * set of AS`Name
  -> set of AS`Name
getFuncAppMapCampExpr(expr,sn) ==
  getFuncAppMaplet(expr.elem,sn)
  union dunion {getFuncAppMultBind(b,sn) | b in set elems expr.bind}
  union if expr.pred = nil then {} else getFuncAppExpr(expr.pred,sn);

getFuncAppRecordModification : AS`RecordModification
  * set of AS`Name -> set of AS`Name
getFuncAppRecordModification(m,sn) ==
  getFuncAppExpr(m.new,sn);

getFuncAppBind : AS`Bind * set of AS`Name -> set of AS`Name
getFuncAppBind(bind,sn) ==
  cases bind:
```

```
  mk_AS`SetBind(pat,Set,-) -> getFuncAppPattern(pat,sn)
    union getFuncAppExpr(Set,sn),
  mk_AS`TypeBind(pat,-,-)  -> getFuncAppPattern(pat,sn)
  end;

getFuncAppMultBind : AS`MultBind * set of AS`Name -> set of AS`Name
getFuncAppMultBind(bind,sn) ==
  cases bind:
  mk_AS`MultSetBind(pats,Set,-)
    -> dunion {getFuncAppPattern(p,sn) | p in set elems pats}
      union getFuncAppExpr(Set,sn),
  mk_AS`MultTypeBind(pats,-,-)
    -> dunion {getFuncAppPattern(p,sn) | p in set elems pats}
  end;

getFuncAppPattern : AS`Pattern * set of AS`Name -> set of AS`Name
getFuncAppPattern(pat,sn) ==
  cases pat:
  mk_AS`PatternName(-,-,-) -> {},
  mk_AS`MatchVal(e,-) -> getFuncAppExpr(e,sn),
  mk_AS`SetEnumPattern(els,-)
    -> dunion {getFuncAppPattern(e,sn) | e in set elems els},
  mk_AS`SetUnionPattern(l,r,-)
    -> getFuncAppPattern(l,sn) union getFuncAppPattern(r,sn),
  mk_AS`SeqEnumPattern(els,-)
    ->  dunion {getFuncAppPattern(e,sn) | e in set elems els},
  mk_AS`SeqConcPattern(l,r,-)
    -> getFuncAppPattern(l,sn) union getFuncAppPattern(r,sn),
  mk_AS`TuplePattern(fields,-)
    -> dunion {getFuncAppPattern(e,sn) | e in set elems fields},
  mk_AS`RecordPattern(-,fields,-)
    -> dunion {getFuncAppPattern(e,sn) | e in set elems fields},
  others -> {}
  end;

\end{vdm_al}
Verifies if an application is of a certain class/module.
\begin{vdm_al}

OtherClassApp : AS`Name * AS`Name -> bool
OtherClassApp(clnm,fnm) ==
  if len fnm.ids > 1
  then
    if fnm.ids(1) = clnm.ids(1)
    then false
    else true
  else false;

operations

\end{vdm_al}
```

```
Removes the functions that can be identified as non recursive out of
the recursion map.
\begin{vdm_al}

removeNonRec : map AS`Name to ENV`FunctionInfo
  ==> map AS`Name to ENV`FunctionInfo
removeNonRec(recmap) ==
  let recmap' =
    { x |-> recmap(x) | x in set dom recmap & recmap(x).rec <> {}},
      recmap'' =
    { x |-> mk_ENV`FunctionInfo(
      { y | y in set recmap'(x).rec & y in set dom recmap'},
     recmap(x).meas,recmap(x).printPO)    | x in set dom recmap'}
  in if recmap'' = recmap then return recmap
     else removeNonRec(recmap'');

\end{vdm_al}
This function, builds the recursion map based on the functions definition of a
class/module. It first starts by finding out the functions applications in each
function and then build a first version of the recMap. After this all the function
calls that can't be found on this class are searched and the same process is
made until the map is closed. This is, elements in the range are all in the domain
too. When this all process is finished, the definitly non recursive functions are
removed and the transitive closure is made to find out the recursive functions
and then the cleaning is again made.
\begin{vdm_al}

BuildRecMap : AS`Name * map AS`Name to AS`FnDef  ==>
              map AS`Name to ENV`FunctionInfo
BuildRecMap(clnm,fnm) ==
  ( dcl recMap : map AS`Name to ENV`FunctionInfo := {|->};
    for all nm in set dom fnm
    do
    (
      let allfns' = dunion { dom x.fcts | x in set rng
#ifdef VDMPP
      VCM`GetParseEnv()
#endif VDMPP
#ifdef VDMSL
      VCM`GetModuleEnv()
#endif VDMSL
      },
          allapps'= getFuncAppFnDef(fnm(nm),{}),
          allfns  = { DestroyCid(x) | x in set allfns'},
          allapps = { DestroyCid(x) | x in set allapps' },
          normout' = { x | x in set allapps &
                           OtherClassApp(clnm,x)},
          normout = normout' inter allfns,
          inapps  = allapps \ normout,
          normin = {NormalizeName(clnm,x) | x in set inapps} inter allfns
        -- normin  = { DestroyCid(x) | x in set normin' },
```

```
          --normout = { DestroyCid(x) | x in set outapps }
      in
        recMap := recMap ++ {mk_AS`Name(clnm.ids^nm.ids,0) |->
                             mk_ENV`FunctionInfo(normin union normout,
                                                 if is_AS`ExplFnDef(fnm(nm))
                                                 then fnm(nm).measu
                                                 else nil,
                                                 false)};
    );
    -- at this point recMap contains all functions associated with
    -- their calls of this class what now is needed is to find the
    -- calls of functions that are mentioned that don't belong to this class.
    recMap := recConstructMap(dunion {x.rec | x in set rng recMap}
       \ dom recMap ,dom recMap,recMap);
    recMap := removeNonRec(recMap);
    recMap := main_RecMap(recMap);
    recMap := removeNonRec(recMap);
    return recMap

  );

\end{vdm_al}
This function finds the function applications in functions that are
not in the class currently beeing type checked. The applications
are found and added to the recMap so that the recursive functions
can be correctly identified
\begin{vdm_al}

recConstructMap : set of AS`Name * set of AS`Name
  * map AS`Name to ENV`FunctionInfo
    ==> map AS`Name to ENV`FunctionInfo
recConstructMap(toVisit,visited,recMap) ==
  if toVisit = {}
  then return recMap
  else
    let x in set toVisit
    in
      let xapps = GetAppsFn(x),
          newvis = visited union {x}
      in return recConstructMap(
                 (toVisit union xapps.#1) \ newvis,
                 newvis,
                 recMap ++ {x |->
                   mk_ENV`FunctionInfo(xapps.#1,xapps.#2,false)}
                 );

\end{vdm_al}
Normalize names is to transform a name, depending if it as
a class/module on the name, then it remains the same.
Otherwise the class/module name is appended to the name.
\begin{vdm_al}
```

```
NormalizeName : AS`Name * AS`Name ==> AS`Name
NormalizeName(clnm,mk_AS`Name(f,cid)) ==
 if clnm.ids = [ f(i) | i in set inds f & i <= len clnm.ids]
 then return mk_AS`Name(f,cid)
 else return mk_AS`Name(clnm.ids^f,cid);

\end{vdm_al}
Destroying the context is needed to use the set "belong"
\begin{vdm_al}
DestroyCid : AS`Name ==> AS`Name
DestroyCid(nm) ==
  return mk_AS`Name(nm.ids,0);

\end{vdm_al}
Gets the names that are in a functions definition. These names
should be intersected with the whole function names in order to
find out which ones are functions and the ones that aren't.
\begin{vdm_al}
GetAppsFn : AS`Name ==> (set of AS`Name) * [AS`Name]
GetAppsFn(nm) ==
  let f = VCM`GetFuncDefWOCtxt(nm)
  in
    if f <> nil
    then return mk_(getFuncAppFnDef(f,{}),f.measu)
    else return mk_({},nil)
pre len nm.ids > 1;


end REC

\end{vdm_al}

\subsection{Test Coverage}

\begin{rtinfo}[MostGeneralNumericType]
{rtinfo.ast}[REC]
\end{rtinfo}
```

List of Figures