Automatic Test Data Generation From VDM-SL Specifications

A dissertation submitted at The Queen's University of Belfast

by

Richard Atterer

April 7, 2000

Acknowledgements

I would like to thank my supervisor, Dr Ivor Spence, for his help and support, as well as for giving me a demanding, but also very interesting task.

This project is based on previous work by Christophe Meudec, and would not have been possible without the theoretical foundations provided by his Ph.D. thesis.

Additionally, I want to express my thanks to the Faculty of Computer Science at the Technische Universität München and especially Mrs Angelika Reiser for letting me study abroad in Belfast for one year.

Abstract

Testing is an important aspect of software development and plays a major role in detecting errors in implementations. Tests are often performed manually and at random, which is problematic because it is time-consuming and there is no way of telling how well a software component has been tested. Additionally, the "random" test inputs are determined by a human, which can mean that the corner case that has escaped the implementor's attention may well also not be noticed by the tester.

It seems obvious that testing should also (at least in part) be done by computers. However, this proves difficult when only the program source is available, as any tool analysing it will hardly be able to tell what the code *is supposed* to do. For this reason, one approach to automatic test data generation is not to use the source code, but an additional, high-level and abstract specification which describes the behaviour of a component. The program developed for this project is a partial implementation of such a test data generation tool.

Contents

1 Introduction

The Vienna Development Model (VDM) and its specification language (VDM-SL) provide means for software developers to ensure the software they create is of very high quality, containing far fewer bugs than with more conventional software development methods. This is achieved by giving a high-level, possibly implicit specification of *what* each component does rather than *how* it is implemented, and giving a formal proof that the behaviour of a component does not change when implicit expressions are replaced with a more concrete, executable version.

Unfortunately, it is impossible to account for all circumstances which can lead to incorrect behaviour of an implementation, so tests of the final software are as important as for programs developed less rigorously.

With conventionally developed programs, testing is mostly performed manually, not very rigorously and takes up a significant amount of the total development cost – without there being any guarantee that the tests even approach to cover all possible cases.

On the other hand, with the high-level specification of programs developed using VDM, it is possible to some degree to automatically generate input samples for each software component. Even though the technique used is largely based on heuristics, it is far superior to manual testing in that the tests are more likely to contain critical cases. Additionally, no human interaction is required to perform tests, making development cheaper.

The program developed for this Software Engineering Project implements a test data generator, based on the algorithm suggested in [Meudec98].

2 Specification

The aim of this Software Engineering Project is to produce a program which, given a formal specification in the VDM specification language (in the form of an ASCII text file), performs an analysis of certain parts of the specification and generates data which can subsequently be used to test an implementation of that specification.

This is to be achieved in three major stages: First, the stream of ASCII characters is turned into a stream of tokens by a lexical analyser (scanner). Next, a parser uses the tokens to build a tree representation of the language constructs. Finally, this tree is traversed and analysed to produce the test data.

Since the program is a command line utility, its user interface is very simple. Command line switches allow the user to influence its behaviour, e.g. `--partition` to output an intermediate result of the test generation process in addition to the test data.

Here is a VDM-SL specification with an example for a function that the program can process:

```
functions
  example(x: int) r: bool   -- integer argument x, boolean return value r
  pre     x <> 5            -- the function will never be called with x = 5
  post    r <=> (x > 1)     -- return true if x greater than 1
```

The creation of test data during the final stage constitutes the most important part of the program. The fundamental idea behind the algorithm is to provide sets of values for the arguments given to a function (and for what it returns) in such a way that each set of values, i.e. each test case, represents one particular "sub-case" of the problem that the function is supposed to handle, and also that all test cases taken together represent the problem as a whole.

Each test case's set of variable values[1] is correct in the sense that it does not violate the specified pre- and post-conditions: If the pre-condition states $x \neq 5$ then none of the generated test cases will assign the value 5 to x. This does not only apply to pre/post-conditions, but also to other constraints, e.g. those imposed by type invariants (although the program only supports the basic data types bool and int, which do not have invariants).

There is a strong similarity between the way the problem is described by the set of all test cases and the way a function with boolean input/output can be described using its disjunctive normal form (DNF). In fact, if only boolean variables and operators are used, the two are the same; joining the test cases using logical or operations yields the DNF.

[1] In this and the following paragraphs, "variable" refers to both the function arguments and the return value.

To make the test case generation possible, it is first necessary to create a *partition* for the expression representing a function. The term "partition" is used to emphasize that the *definition domain* – for example, $\mathbb{Z} \times \mathbb{B}$ for one integer argument and a boolean return value – is subdivided into disjoint parts (*definition domain subsets* or *sub-partitions*), each of which is considered to be independent from the others for the purpose of testing, so that only one sample variable value needs to be taken from each. For example, the function's pre-condition $x \neq 5$ leads to a subdivision of $x \in \mathbb{Z}$ into the sub-partitions $x < 5$ and $x > 5$.

For the example above, the expression is $x \neq 5 \wedge r \Leftrightarrow (x > 1)$, and the corresponding partition could be written as

$$\left\{ \begin{array}{c} x < 5 \\ x > 5 \end{array} \right\} \times \left\{ \begin{array}{c} \{r\} \times \{x > 1\} \\ \{\neg r\} \times \{x \leq 1\} \end{array} \right\}$$

With this notation, sub-partitions are enclosed in $\{\}$ if their union is the whole definition domain . The '$\times$' operator is used to connect the domains represented by two sub-expressions if both of them must be true at the same time. From a more abstract point of view, the operator performs an intersection of its left-hand and right-hand domain arguments.

A test data generation program creates the individual variable values by combining sub-partitions into a predicate (also called *equivalence class* in [Meudec98]) and then solving that. If there is more than one possibility for a variable value, one of the possible values is chosen at random, since any value represents its sub-domain equally well. For the above example $x \neq 5$, the first sub-partition, $x < 5$, might lead to x being assigned the value -46 for a first test case, and the second one, $x > 5$, to a value of 7 for a second test case, or indeed any other value greater than 5.[2]

If sub-partitions have been created for several parts of the original expression, the only way to ensure that all aspects of the problem are taken into account is to combine these sub-partitions in all possible ways, and to create one test case for each of the permutations. In the example, there are two subdivisions of the domain of x: The pre-condition subdivides $\mathbb{Z}$ into $x < 5$ and $x > 5$, and the post-condition leads to the generation of the sub-partitions $r \wedge x > 1$ and $\neg r \wedge x \leq 1$. Combining the sub-partitions in all possible ways results in four equivalence classes: $x < 5 \wedge r \wedge x > 1$, $x > 5 \wedge r \wedge x > 1$, $x < 5 \wedge \neg r \wedge x \leq 1$ and $x > 5 \wedge \neg r \wedge x \leq 1$. The last of these predicates is contradictory, so no test cases are created for it.

To summarize, the program must perform the following tasks:

[2] In practice, the program will use the four sub-partitions $x < 4$, $x = 4$, $x = 6$, $x > 6$ to catch corner cases.

- Read the input characters and separate them into symbols in the scanner.
- In the parser, analyse these symbols and build a parse tree which represents the VDM-SL specification.
- In the parse tree, find the function/operation definition that is to be analysed and create a partition from the expressions it contains.
- Create equivalence classes from the partition and output them.

3 Design

The program is written in the C++ programming language and makes use of its standard library as described in [Stroustrup97]. It is expected that the VDM-SL input given to it is syntactically and semantically correct – however, the program gives an error message and exits for all those cases of invalid input that do not allow it to continue processing.

3.1 Lexical Analysis

The scanner is supplied a filename on the command line and reads characters from the specified file, grouping them together and either ignoring them (comments and whitespace) or passing them on to the parser as terminal tokens.

Because many symbols used in VDM-SL are not available with ASCII the *Interchange Concrete Syntax* as defined in section 10 of [ISO93] is used. The code of the scanner is generated with the lex or flex tool from a description file, which needs to be written with care to deal with the following issues:

- Provide regular expressions in such a way that VDM-SL language constructs as described by the ISO standard are recognized. Reject with an error message any character sequences that cannot form part of VDM-SL language constructs. Silently process comments, without passing on any information about them.
- Where necessary, e.g. for integer or string constants, process the recognized characters and pass the additional information to the parser.
- Maintain a count of the current line number, for use with error and warning messages. Along with each terminal token (whether consisting of just a single character or of several characters), also pass the current line number to the parser.

Data Model and Functions

The scanner code (a function called yylex()) is generated from the description file which is passed to the lex or flex utility.

The scanner maintains a hash table in which it stores a pointer to each identifier it encounters.

3.2 Parsing

From the terminal tokens, a tree structure is built by means of a parser. This task is performed by a program generated by the yacc or bison tool after the grammar of VDM-SL described in section 9 of [ISO93] has been transformed into the LR(1) grammar accepted by it. The operator precedence rules from section 9.8 are incorporated with special yacc/bison extensions.

The grammar transformation is made very difficult by the fact that the supplied grammar is not LR(1) and contains numerous ambiguities and errors, all of which need to be resolved to make the parser work. It is likely that many of them will have been corrected in the final version of the ISO VDM-SL standard, but that version is not available for free.

Even though later stages of the program only need to deal with *parts* of the VDM-SL source, the program recognizes *all* of the language (with the exception of statements) and provides a tree representation for it, as it is intended to provide a basis for further projects dealing with VDM-SL.

Data Model and Functions

The part of the program which builds a tree out of the terminal tokens can be subdivided into the grammar description for the parser generator, the parser code generated from it (consisting mainly of a function yyparse()) and the data structures for the parse tree.

In order to ease debugging and extending the program, the tree is not represented by instances of a single "node" type. Instead, advantage is taken of the C++ type checking mechanism by providing a separate class definition for each nonterminal rule in the grammar. The class name (declared in the Tree namespace) is very similar to the name of the nonterminal rule, e.g. class ExplicitFunctionDefinition for the rule *explicit-function-definition*.

A hierarchy of classes is derived from the virtual base class for a nonterminal token, class Tree::N. If one of the classes is only intended for use as a base class, it is made abstract – for example, this is the case for class N or for class FunctionDefinition, from which ExplicitFunctionDefinition and ImplicitFunctionDefinition derive.

The constructor of each class takes arguments in the order in which the respective subterminals or nonterminals appear in the grammar rule. Whenever one of these symbols is optional in the grammar and is not present in the current input, this is indicated by passing a null pointer to the constructor.

The destructor of each class deletes all subtrees and symbols they contain, with one exception: Identifiers are not deleted because other pointers to them exist in the hash table.

Copying and assigning tree node objects is disallowed.

After the object has been constructed, references to the subtrees of a nonterminal node can

be obtained through calls to member functions whose names are closely related to the type of tree they return, e.g. ExplicitFunctionDefinition::preCondition() for the expression representing the pre-condition of a function. If the sub-symbol is optional, the caller must first check for its presence with, e.g., hasPreCondition().

A nonterminal representing a list *is a* (derives publicly from the) standard library vector.

The parser supports the concept of name scopes, since knowledge about where variables are defined must be accessible in order to correctly determine whether identifiers in pre-conditions refer to "old" variable values even though they are spelled without a trailing ˜.

3.3 Generation and Output of Test Data

Once the tree of a VDM-SL specification has been created, it is analysed according to the algorithms explained in chapter 4 and 5 of [Meudec98]: For the function that the user has indicated, the pre-condition and post-condition are examined, and a partition representing them is generated. Subsequently, the partition is used to produce predicates which need to be passed to an external solver to find out whether they are satisfiable, and, if they are, to choose a random sample from each partition. The predicate expressions are printed to the screen or to a file.

It is assumed that it is possible for the tester not only to pass certain parameters to a function when performing tests on an implementation, but also to instantiate a given state in advance, e.g. by modifying global variables in the program.

The test data generation tool does not address the problem of data type transformations, such as the necessity of storing the members of a set in a particular order in any implementation of a specification.

Although the algorithm used is based on that described in [Meudec98], this implementation is slightly different from the one proposed in chapter 5.3 of that work. The following sections explain the differences and give justifications for the changes.

3.3.1 Processing the Input

In sections 5.3.1 and 5.3.4 of his thesis, Meudec describes his idea of using different parsers to process the input tokens differently depending on the operators by which expressions are combined into larger expressions. In practice, this is not feasible because the available parser generator tools only have one lookahead token, which, in the case of infix operators, makes it impossible to switch over to the correct parser before the left-hand operand is processed.

One possible solution for this problem would be to make multiple passes over the input tokens, but a preferable way of achieving the same result is to build a "conventional" parse tree first and create the operator-dependent information (which describes the partitions of the expression) during a second pass over this tree.

Apart from being more efficient, this also allows for moving the partitioning code to a different compilation unit, to stress the fact that parsing and partitioning are two major different parts of the program. Finally, experience shows that having the parse tree available for later inspection is essential for all but very simple compiler-like programs.

3.3.2 Partitioning Expressions

The process of generating the nested partition description is performed as outlined in section 2, with minor differences to [Meudec98, 5.3.4, p.134].

The term "nested partition description" is to be understood as: None of the equivalence class "full combination" operators (denoted by the symbol '$\times$') have yet been applied to any of their arguments – in other words, the partition description tree has not been flattened.

The behaviour of the program differs from the algorithm proposed by Meudec in the following ways:

- Whereas Meudec's algorithm suggests the use of parsers, this program uses a set of possibly recursive methods, each one associated with a different node type of the parse tree.
- Instead of providing ... True and ... Undef along with a Negate function, the program uses partTrue(), partFalse() and partUndef() functions – obviously, this does not make a difference, except that a call partFalse(x) instead of partTrue(Negate(x)) leaves less work to any expression simplification or solving which might take place later on.
- There are no separate Coarse and Refine versions of the partitioning rules/code. This is not necessary because closer examination shows that the distinction is only conceptual (Coarse for path expressions, Refine for path and non-path expressions)[3] and that the subset of the Refine partitioning rules applying to path sub-expressions is identical to the Coarse partitioning rules.
- In [Meudec98, 4.2.2, p.93], the author remarks that it makes sense to provide partitions not only for non-logical expressions, but also for expressions with a non-boolean result. In particular, the possibility of division by zero should be addressed. This is implemented

[3] [Meudec98, 4.1.1, p.73] defines *path expressions* to denote if and cases expressions and logical expressions.

using a partDef() method instead of the two methods partTrue() and partFalse() for all operators whose result is not boolean.

It seems appropriate to introduce a more flexible notation to express combinations, with more options than just '$\times$' for "all combinations" – from now on, this dissertation will use a bipartite graph as an infix operator between two sets of equivalence classes to indicate how the classes should be combined. For example,

$$\left\{ \begin{array}{c} a \\ b \\ c \end{array} \right\} \bowtie \left\{ \begin{array}{c} d \\ e \\ f \end{array} \right\} \qquad \text{is equivalent to} \qquad \left\{ \begin{array}{c} a \\ b \\ c \end{array} \right\} \times \left\{ \begin{array}{c} d \\ e \\ f \end{array} \right\}$$

and

$$\left\{ \begin{array}{c} a \\ b \\ c \end{array} \right\} \diagup\!\diagdown \left\{ \begin{array}{c} d \\ e \\ f \end{array} \right\} \qquad \text{is equivalent to} \qquad \left\{ \begin{array}{c} a \wedge d \\ a \wedge e \\ c \wedge d \\ c \wedge e \\ c \wedge f \end{array} \right\}$$

The distributed union operator, $\bigcup\{\ldots\}$, is (incorrectly) omitted for the sake of a more concise notation.

The program written for this project only deals with the following VDM-SL operators: The logical operators not, and, or, =>, <=>, the relational operators =, <>, >, <, >=, <= and the arithmetic operators +, -, *, div. Variables can be of the types bool and int only.[4]

Here are the partitioning rules for the operators. They are basically the same as Meudec's, but have been altered in accordance to the modifications described above. All e_i denote sub-expressions.

The behaviour of the logical operators is determined by VDM's three-valued logic, see [Meudec98, table 4.1, p.85]:

$$\mathsf{partTrue}(\mathsf{not}\ e_1) = \mathsf{partFalse}(e_1)$$

$$\mathsf{partFalse}(\mathsf{not}\ e_1) = \mathsf{partTrue}(e_1)$$

$$\mathsf{partUndef}(\mathsf{not}\ e_1) = \mathsf{partUndef}(e_1)$$

$$\mathsf{partTrue}(e_1\ \mathsf{and}\ e_2) = \mathsf{partTrue}(e_1) \times \mathsf{partTrue}(e_2)$$

[4] Notice that this simplification implies that invariants of user-defined types never need to be taken into account.

$$\mathsf{partFalse}(e_1 \text{ and } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \\ \mathsf{partFalse}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partTrue}(e_2) \\ \mathsf{partFalse}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

$$\mathsf{partUndef}(e_1 \text{ and } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partTrue}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

$$\mathsf{partTrue}(e_1 \text{ or } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \\ \mathsf{partFalse}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partTrue}(e_2) \\ \mathsf{partFalse}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \text{ or } e_2) = \{\mathsf{partFalse}(e_1) \times \mathsf{partFalse}(e_2)\}$$

$$\mathsf{partUndef}(e_1 \text{ or } e_2) = \left\{ \begin{array}{c} \mathsf{partFalse}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partFalse}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

$$\mathsf{partTrue}(e_1 \texttt{ => } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \\ \mathsf{partFalse}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partTrue}(e_2) \\ \mathsf{partFalse}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ => } e_2) = \mathsf{partTrue}(e_1) \times \mathsf{partFalse}(e_1)$$

$$\mathsf{partUndef}(e_1 \texttt{ => } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partFalse}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

$$\mathsf{partTrue}(e_1 \texttt{ <=> } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \times \mathsf{partTrue}(e_2) \\ \mathsf{partFalse}(e_1) \times \mathsf{partFalse}(e_2) \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ <=> } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \times \mathsf{partFalse}(e_2) \\ \mathsf{partFalse}(e_1) \times \mathsf{partTrue}(e_2) \end{array} \right\}$$

$$\mathsf{partUndef}(e_1 \texttt{ <=> } e_2) = \left\{ \begin{array}{c} \mathsf{partTrue}(e_1) \\ \mathsf{partFalse}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partTrue}(e_2) \\ \mathsf{partFalse}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

The rules for the relational operators are not "set in stone" like those for the logical operators above. Instead, they are heuristics which aim at finding the cases where errors are most probable in an implementation. The following rules suggested in [Meudec98, p.139ff] are likely to catch corner cases. Note that the $\mathsf{partUndef}(\dots)$ rules can be made slightly simpler than Meudec's version because of the assumption that the input to the program is semantically correct. $\mathsf{partUndef}(e_1 \circ e_2)$ is identical for all $\circ \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{=}, \texttt{<>}\}$:

$$\mathsf{partUndef}(e_1 \circ e_2) = \left\{ \begin{array}{c} \mathsf{partDef}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \times \left\{ \begin{array}{c} \mathsf{partDef}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\}$$

$$\mathsf{partTrue}(e_1 \texttt{ < } e_2) = \mathsf{partDef}(e_1) \times \mathsf{partDef}(e_2) \times \left\{ \begin{array}{c} e_1 + 1 = e_2 \\ e_1 + 1 < e_2 \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ < } e_2) = \mathsf{partTrue}(e_1 \texttt{ >= } e_2)$$

$$\mathsf{partTrue}(e_1 \texttt{ > } e_2) = \mathsf{partDef}(e_1) \times \mathsf{partDef}(e_2) \times \left\{ \begin{array}{c} e_1 = e_2 + 1 \\ e_1 > e_2 + 1 \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ > } e_2) = \mathsf{partTrue}(e_1 \texttt{ <= } e_2)$$

$$\mathsf{partTrue}(e_1 \texttt{ <= } e_2) = \mathsf{partDef}(e_1) \times \mathsf{partDef}(e_2) \times \left\{ \begin{array}{c} e_1 = e_2 \\ e_1 < e_2 \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ <= } e_2) = \mathsf{partTrue}(e_1 \texttt{ > } e_2)$$

$$\mathsf{partTrue}(e_1 \texttt{ >= } e_2) = \mathsf{partDef}(e_1) \times \mathsf{partDef}(e_2) \times \left\{ \begin{array}{c} e_1 = e_2 \\ e_1 > e_2 \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ >= } e_2) = \mathsf{partTrue}(e_1 \texttt{ < } e_2)$$

$$\mathsf{partTrue}(e_1 \texttt{ = } e_2) = \mathsf{partDef}(e_1) \times \mathsf{partDef}(e_2) \times \left\{ \begin{array}{c} e_1 = e_2 \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ = } e_2) = \mathsf{partTrue}(e_1 \texttt{ <> } e_2)$$

$$\mathsf{partTrue}(e_1 \texttt{ <> } e_2) = \mathsf{partDef}(e_1) \times \mathsf{partDef}(e_2) \times \left\{ \begin{array}{c} e_1 = e_2 + 1 \\ e_1 > e_2 + 1 \\ e_1 + 1 = e_2 \\ e_1 + 1 < e_2 \end{array} \right\}$$

$$\mathsf{partFalse}(e_1 \texttt{ <> } e_2) = \mathsf{partTrue}(e_1 \texttt{ = } e_2)$$

The arithmetic operators +, - and * simply combine the partitions of their arguments, whereas division by zero necessitates some additional possibilities for that operator. Again, because the input specification is assumed to be semantically correct, certain cases which would lead to the expression becoming undefined can be omitted. For all $\circ \in \{\texttt{+}, \texttt{-}, \texttt{*}\}$:

$$\mathsf{partDef}(e_1 \circ e_2) = \mathsf{partDef}(e_1) \times \mathsf{partDef}(e_2)$$

$$\mathsf{partUndef}(e_1 \circ e_2) = \left\{ \begin{array}{c} \mathsf{partDef}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \bigtimes \left\{ \begin{array}{c} \mathsf{partDef}(e_2) \\ \mathsf{partUndef}(e_2) \end{array} \right\}$$

$$\mathsf{partDef}(e_1 \texttt{ div } e_2) = \mathsf{partDef}(e_1) \times \mathsf{partTrue}(e_2 \texttt{ <> } 0)$$

$$\mathsf{partUndef}(e_1 \text{ div } e_2) = \left\{ \begin{array}{c} \mathsf{partDef}(e_1) \\ \mathsf{partUndef}(e_1) \end{array} \right\} \bigtimes \left\{ \begin{array}{c} \mathsf{partTrue}(e_2 \text{ <> } 0) \\ \left\{ \begin{array}{c} \mathsf{partTrue}(e_2 \text{ = } 0) \\ \mathsf{partUndef}(e_2) \end{array} \right\} \end{array} \right\}$$

If e_1 is an integer variable or constant, the partitioning is straightforward:[5]

$$\mathsf{partDef}(e_1) = \mathsf{true}$$

$$\mathsf{partUndef}(e_1) = \mathsf{false}$$

For boolean variables or constants, the following must be provided instead of partDef() and partUndef():

$$\mathsf{partTrue}(e_1) = \{e_2\}$$

$$\mathsf{partFalse}(e_1) = \{\text{not } e_2\}$$

The program applies a *dynamic programming* technique; it avoids generating partitions for, say, partTrue(x), more than once: If partTrue(x) is encountered a second time during generation of the nested partition description, the previously generated version is re-used.

As an example, here is the nested partition description generated for the expression $x > 0$ or $y < 5$ in [Meudec98, 5.4.1, p.148].

$$\left\{ \begin{array}{c} \mathsf{true} \times \mathsf{true} \times \left\{ \begin{array}{c} x = 0+1 \\ x > 0+1 \end{array} \right\} \\ \mathsf{true} \times \mathsf{true} \times \left\{ \begin{array}{c} x = 0 \\ x < 0 \end{array} \right\} \\ \left\{ \begin{array}{c} \mathsf{true} \\ \mathsf{false} \end{array} \right\} \bigtimes \left\{ \begin{array}{c} \mathsf{true} \\ \mathsf{false} \end{array} \right\} \end{array} \right\} \bigtimes \left\{ \begin{array}{c} \mathsf{true} \times \mathsf{true} \left\{ \begin{array}{c} y+1 = 5 \\ y+1 < 5 \end{array} \right\} \\ \mathsf{true} \times \mathsf{true} \left\{ \begin{array}{c} y = 5 \\ y > 5 \end{array} \right\} \\ \left\{ \begin{array}{c} \mathsf{true} \\ \mathsf{false} \end{array} \right\} \bigtimes \left\{ \begin{array}{c} \mathsf{true} \\ \mathsf{false} \end{array} \right\} \end{array} \right\}$$

The obvious simplifications of eliminating true and false are performed by the program during the building of the partition, so the result will correspond to:

$$\left\{ \begin{array}{c} \left\{ \begin{array}{c} x = 0+1 \\ x > 0+1 \end{array} \right\} \\ \left\{ \begin{array}{c} x = 0 \\ x < 0 \end{array} \right\} \end{array} \right\} \bigtimes \left\{ \begin{array}{c} \left\{ \begin{array}{c} y+1 = 5 \\ y+1 < 5 \end{array} \right\} \\ \left\{ \begin{array}{c} y = 5 \\ y > 5 \end{array} \right\} \end{array} \right\}$$

[5] The reason why true and false are not enclosed in {} here is that no Tree::Expr or Part::Partition object is generated for them, in contrast to everything inside {}. Instead, elimination of this special true/false takes place immediately during the tree construction.

However, the implementation does not have a representation for the partial combination operator, consequently its three different possibilities are enumerated in a partition set, the final output being:

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} x=0+1 \\ x>0+1 \end{array} \right\} \times \left\{ \begin{array}{l} y+1=5 \\ y+1<5 \end{array} \right\} \\ \left\{ \begin{array}{l} x=0+1 \\ x>0+1 \end{array} \right\} \times \left\{ \begin{array}{l} y=5 \\ y>5 \end{array} \right\} \\ \left\{ \begin{array}{l} x=0 \\ x<0 \end{array} \right\} \times \left\{ \begin{array}{l} y+1=5 \\ y+1<5 \end{array} \right\} \end{array} \right\}$$

Data Model and Functions

partTrue(), partFalse(), partDef() and partUndef() can be called in several ways: First, each Tree::Expr[6] class provides virtual member functions of that name, so that a call obj.partDef() (without any arguments) can be used to generate the partition for an expression "obj". However, the member functions just make calls to normal functions in the Part namespace, for example Part::Def::less(Tree::Expr*, Tree::Expr*). This system allows both for using the operator and subexpression(s) stored in an Expr object and for supplying two subexpressions to functions for specific operators. The latter is useful for cases like partFalse(e_1 = e_2) = partTrue(e_1 <> e_2), where an object for the <> would otherwise have to be created just to make the partitioning call.

The nested partition description is represented by a tree of nodes whose classes are derived from the abstract base class Part::Partition. There are five such classes:

- FullComb represents the full combination operator and consists of two sub-partitions which are to be combined in all possible permutations.
- PartSet is a set of equivalence classes – several sub-partitions enclosed in $\{\}$ in the description above. The effect of the partial combination "bipartite graph" operator is achieved in the program by enumerating the possibilities in a PartSet.
- ExprSet is equivalent to PartSet, except that it contains pointers to Tree::Expr objects instead of further Partition objects. Thus, it represents the leaves of the partition tree.
- ConstTrue and ConstFalse are used to denote an ExprSet with just one entry which is the boolean constant true or false. In contrast to ExprSet, however, they are eliminated

[6] Tree::Expr is derived from Tree::N and is the abstract base class of all nodes which represent expressions.

during partition generation, so they will not occur in the final partition unless that partition evaluates to $\{\mathsf{true}\}$ or $\{\mathsf{false}\}$. Only one instance is ever created for each class.

The Partition classes have additional member functions, which are described later on.

3.3.3 Generation of Equivalence Classes

Now that the partition description is available, a set of expressions must be created, each representing one test case. This is where the program differs most from [Meudec98].

The technique used by Meudec is as follows: Recursively replace all combination operators with a set of equivalence classes:

1. For each combination operator, first the left and right operand are turned into sets of equivalence classes (or predicate expressions) by means of recursive application of the technique.

2. The new equivalence classes are generated by picking one expression from the left-hand and one from the right-hand set argument for each possible permutation, and applying a logical and to the expressions.
   If the left and right argument are independent from each other in the sense of the description in [Meudec98, 5.3.3, p.133], not all permutations are tried. Instead, the algorithm only ensures that each element of the left and each element of the right operand has been picked at least once.

3. The new, larger equivalence class is passed to a solver which eliminates contradictory cases.

It must be noted that this approach contains a flaw: Suppose a partition

$$\left\{ \left\{ \begin{array}{c} a \\ b \end{array} \right\} \bigtimes \left\{ \begin{array}{c} c \\ d \end{array} \right\} \right\} \times e$$

is to be reduced to equivalence classes, and the arguments of the bipartite graph operator are independent. Meudec's algorithm *might* (it is non-deterministic) decide to drop the permutation $a \wedge c$ and turn this into

$$\left\{ \begin{array}{c} a \wedge d \\ b \wedge c \end{array} \right\} \times e$$

On the other hand, it might also turn out to be

$$\{\} \times e$$

because both $a \wedge d$ and $b \wedge c$ are contradictory, whereas the discarded $a \wedge c$ may have been soluble. *This* is easily correctable: Just check which expressions cannot be solved and if any of a, b, c or d is eliminated completely, try the discarded permutations as a last resort.

However, assuming that $a \wedge d$ and $b \wedge c$ do have solutions, the following final partition would be generated:

$$\left\{ \begin{array}{l} a \wedge d \wedge e \\ b \wedge c \wedge e \end{array} \right\}$$

At this point, it is possible that both $a \wedge d$ and $b \wedge c$ are soluble, but that neither $a \wedge d \wedge e$ nor $b \wedge c \wedge e$ is soluble, whereas $a \wedge c \wedge e$ could have been soluble – and this may happen at any level of depth, so it is impossible to account for.

To summarize, by throwing away certain combinations early, the algorithm may fail to generate important test cases later on. In the worst case, it might generate no test cases at all – and even worse, an implementation could run for a very long time before this happens.

The algorithm used by the program has several advantages over Meudec's technique even though it does not take independence into account at all. It has disadvantages that may render it as inapplicable to large problems as the original algorithm, but is believed to get closer to an application that can be used in practice, i.e. for large input specifications.

The changes were motivated not only by the flaw in the first technique, but also by the following consideration: Whatever algorithm is used, passing large expressions will cause the program to run for a very long time (maybe months) because of the exponential explosion of the number of combinations. In this light, it is completely unacceptable to first generate all equivalence classes and then output them – not only will it take too long before they are output, they will also all have to be stored in memory at one point, which might be impracticable because of the size of the data.

Consequently, the algorithm should not be in O(NP) for both space and time – and while polynomial complexity for time cannot be achieved, this is possible for the space requirements. Furthermore, even if the program runs for months, the *first* equivalence class should be output (almost) immediately, so that testing can take place in parallel with the test data generation. Because it may well be impossible to wait until all test cases have been generated, an additional requirement is that two subsequent test cases should be as different from each other as possible. By swapping cases semi-randomly, the chances are higher that if the program is terminated prematurely, the equivalence classes generated so far represent the problem reasonably well. Moreover, the semi-random swapping avoids cases of the program running into a branch of the partition

where all generated expressions turn out to be contradictory – making the output of valid test cases stop for some (possibly quite long) time.

The basic idea is outlined by the following paragraphs. Notice that in contrast to the solution proposed in [Meudec98], the program only works with the initial nested partition description; the tree representing it is not modified by the algorithm.

It seems that this approach only allows for *either* semi-random generation of test cases *or* taking independence relations between partitions into account, or it will suffer from the same problem as Meudec's version! Since it is expected that for large problems the program will be terminated after only a small fraction of all combinations has been generated, semi-random generation seems to be more important.

1. For each sub-partition x, recursively calculate the number of combinations of expressions, $c(x)$, that need to be generated with it. For a full combination operation on the partitions x_1 and x_2 (which will be sets of partitions in many cases), $c(x) = c(x_1) \cdot c(x_2)$, and for sets of partitions with n members, $c(x)$ is $\sum_{i=1}^{n} c(x_i)$. For the "bipartite graph" combination operator applied to two partition sets l and r, $c(x) = \sum_{i,j} c(l_i) \cdot c(r_j)$ for all combinations indicated by the operator, of the ith partition on the left and the jth partition on the right side.

2. Assign a number in the range $[0 \ldots c(x))$ to each possible permutation of the whole partition. Given that number, it is possible to calculate the particular permutation of sub-partitions associated with the number. In order to achieve the required "shuffling" of the permutations, count a variable upwards from 0 to the next higher power of 2 greater or equal to $c(x)$ – but before calculating the individual permutations from it, mirror the variable value bit by bit. (If the number exceeds $c(x)$ after the bit-mirroring, that combination is ignored.)

3. Turn the "path" through the partition associated with each number into an expression by connecting sub-partitions' expressions with a logical and. The resultant predicate would have to be passed to a solver to determine whether it can be solved, and to generate sample values for the variables if it can. However, since the implementation of a solver is beyond the scope of this project, the program only outputs the predicate.

When this algorithm is applied to the partition of the example $x > 0$ or $y < 5$,

$$\left\{ \begin{array}{c} \left\{ \begin{array}{c} x = 0+1 \\ x > 0+1 \end{array} \right\} \\ \left\{ \begin{array}{c} x = 0 \\ x < 0 \end{array} \right\} \end{array} \right\} \otimes \left\{ \begin{array}{c} \left\{ \begin{array}{c} y+1 = 5 \\ y+1 < 5 \end{array} \right\} \\ \left\{ \begin{array}{c} y = 5 \\ y > 5 \end{array} \right\} \end{array} \right\}$$

then the number of permutations is 12. The following table shows the expressions generated from the partition, both in the original and the "shuffled" order.

ORIGINAL ORDER			SHUFFLED ORDER		
Binary	Nr	Generated expression	Mirrored	Nr	Generated expression
0000	0	$x = 0+1 \wedge y+1 = 5$	0000	0	$x = 0+1 \wedge y+1 = 5$
0001	1	$x > 0+1 \wedge y+1 = 5$	1000	8	$x = 0+1 \wedge y = 5$
0010	2	$x = 0+1 \wedge y+1 < 5$	0100	4	$x = 0 \wedge y+1 = 5$
0011	3	$x > 0+1 \wedge y+1 < 5$	1100	12	–
0100	4	$x = 0 \wedge y+1 = 5$	0010	2	$x = 0+1 \wedge y+1 < 5$
0101	5	$x < 0 \wedge y+1 = 5$	1010	10	$x = 0+1 \wedge y > 5$
0110	6	$x = 0 \wedge y+1 < 5$	0110	6	$x = 0 \wedge y+1 < 5$
0111	7	$x < 0 \wedge y+1 < 5$	1110	14	–
1000	8	$x = 0+1 \wedge y = 5$	0001	1	$x > 0+1 \wedge y+1 = 5$
1001	9	$x > 0+1 \wedge y = 5$	1001	9	$x > 0+1 \wedge y = 5$
1010	10	$x = 0+1 \wedge y > 5$	0101	5	$x < 0 \wedge y+1 = 5$
1011	11	$x > 0+1 \wedge y > 5$	1101	13	–
1100	12	–	0011	3	$x > 0+1 \wedge y+1 < 5$
1101	13	–	1011	11	$x > 0+1 \wedge y > 5$
1110	14	–	0111	7	$x < 0 \wedge y+1 < 5$
1111	15	–	1111	15	–

Unfortunately, compared to the algorithm originally proposed in [Meudec98], the program tends to pass larger expressions to the solver, and more of these predicates are unsatisfiable, which might have a significant impact on performance. A more sophisticated implementation would try to take advantage of both algorithms' positive aspects by proceeding according to Meudec's algorithm first, but switching over to the algorithm used by vdmpart once the size of all sets of equivalence classes in the partition exceeds a certain limit.

Data Model and Functions

Calculating the number of permutations is likely to yield extremely large numbers for non-trivial inputs. Since the 32-bit or 64-bit integers of current hardware are not adequate for all cases, a special concrete type ::BigInt is introduced. The program only implements BigInts using unsigned

long or the non-standard unsigned long long (if supported), but by providing the BigInt abstraction it is made easy to replace this with an arbitrary-length integer implementation later on, should this become necessary.

Most of the functionality of BigInt is implemented by overloading of the respective operators. However, there are some special methods:

- size_t roundUp() rounds the value up to the next power of two, and returns which power of two the BigInt now represents.
- mirrorInc(size_t n) increments what is interpreted to be the bit-mirrored representation of an integer with n bits, i.e. a call to mirrorInc() is equivalent to mirroring the number bit by bit, increasing it by one and mirroring it once more.
- divrem(const BigInt& divisor, BigInt& result, BigInt& remainder) is preferred over the usual integer division and remainder (/ and %) for efficiency reasons; with this method, the division only needs to be carried out once, not twice.

The number of permutations is calculated for each Partition node during its construction and stored with the object. Its value is returned by the combinations() member function.

Partition::partition(BigInt x) is a recursive member function which returns the expression corresponding to a permutation given the permutation's number.

4 Implementation

The program was written in a Unix environment, using the GNU C Compiler, version 2.95, and the tools make, bison, flex, gawk (for the test suite and "`make depend`") and cvs (for version management). The compilation process is made easier by the use of GNU autoconf and the `configure` script it creates. The documentation was created using LaTeX, xfig and lgrind for formatting source code examples.

As the total project size (*ex*cluding generated files) is about 10 000 lines of code, the explanations in this chapter do not attempt to describe it in detail. Instead, they focus on making clear what concepts or algorithms were used for the implementation of each component, and give small examples of the code where appropriate. Furthermore, in an attempt not to confuse with too many details, the text does not even highlight every single aspect of the code excerpts.

4.1 Lexical Analysis

The implementation of the scanner comes in the form of a description file `lex.yy` that is passed to flex. The scanner distinguishes between several types of input tokens, which are passed to the parser in a Symbol<...> object containing the token value as well as the token's position (line number) in the input file. The different types of tokens are integer literals, character literals, string literals, quote literals, identifiers, keywords, operators consisting of more than one character (e.g. **) and single characters.

Whitespace and comments are ignored. The specification of comments is vague in [ISO93]. In particular, it is not clear whether multi-line comments may be nested and how they interact with "--" comments. The implementation ignores annotation or end annotation in single-line "--" comments even if they are inside a multi-line comment, so that -- end annotation never ends a multi-line comment. It allows nesting of multi-line comments. Within a multi-line comment, words are formed in the same way as in the program text, so that x_end annotation or x'end annotation or x#end annotation does not end the comment, but x:end annotation does. On the other hand, character and string literals are *not* treated specially, so any occurrence of annotation inside ' ' or " " is recognized during the processing of multi-line comments.

Here is a typical example of one of the regular expressions in the scanner description file `lex.yy` and its associated action; the part of the file dealing with the recognition of quote literals, i.e. strings of alphabetic characters and '_' enclosed in <>.

```
\<[[:alpha:]][[:alpha:]_]*\>{WHITE} {
    char* scanned = yytext + yyleng - 1;
```

```
  while (*scanned == ' ' || *scanned == '\v' || *scanned == '\t')
    --scanned;
  *scanned = '\0'; // overwrite '>x
  auto_ptr<string> s(new string(yytext + 1));
  if (CmdOptions::Debug::parse)
    cout << "Quote <" << *s << '>' << endl;
  xxlval.str.assign(s.release(), filePos);
  return LIT_QUOTE;
}
```

During the implementation and testing of multi-line comments (annotation ... end annotation) it became obvious that if a user accidentally makes use of the word "annotation" anywhere in the input document, the exact position of the occurrence would be difficult to find if only a "unterminated comment at end of file" error message were given. Consequently, the error message records all line numbers containing annotation or end annotation – see page 47 in section 4.6.2 for an example of the message.

The scanner is responsible for maintaining the hash table hashTab which records the names of all identifiers that are found. Initialisation and clean-up of hashTab is implemented as a "Singleton" class, i.e. a class for which only one instance is ever created during program initialisation and destroyed at program termination. The class definition is given below – the keyword tables have been shortened.

```
struct Singleton_InitHash : private DebugSingleton {
  static const int values[] = {
    MUNION, PSUBSET, SUBSET, DINTER, DUNION, INVERSE, ...
  };
  static const char* const keywords[] = {
    "munion", "psubset", "subset", "dinter", "dunion", "inverse",...
  };
  Singleton_InitHash() {
    for (size_t i = 0; i < sizeof(keywords) / sizeof(char*); ++i)
      hashTab[new IdToken(values[i], keywords[i])];
  }
  ~Singleton_InitHash() {
    Lex::IdToken::allowDelete();
    for (myhashmap::iterator i = hashTab.begin(), e = hashTab.end();
         i != e; ++i)
      delete i->first;
```

```
        Lex::IdToken::denyDelete();
    }
} singleton_InitHash;
```

When an identifier is encountered by the scanner, the following action is taken (see `lex.yy` for the code):

- In the special case of an identifier reading "annotation", change the scanner's mode of operation to skip the comment.
- Check whether the identifier begins with the letters "is_" or "mk_", as the parser requires the scanner to distinguish between these two cases and "normal" identifiers.
- If the identifier begins with the character '$', check whether the part following the '$' is a keyword and give an error if it is not. (See [ISO93, 10.3, p.203])
- Unless the identifier has already been added to hashTab, add it now.
- Return a pointer to the identifier's Lex::IdToken object to the parser, together with the appropriate token type, which is either IDENTIFIER, MK_IDENTIFIER or IS_IDENTIFIER, or another token type (such as OPERATIONS) if the identifier is a keyword.

4.2 Parsing

4.2.1 Creation of the LALR(1) grammar

The parser is implemented in the file `parse.y` which was created by first translating the grammar from [ISO93] into a form that bison can parse, and then altering that representation. This proved to be a very difficult task: From the "reduce/reduce conflict" error messages produced by bison, it can take very long to determine exactly which part of the grammar is incorrect, and, once this is clear, it is sometimes also extremely difficult to correct the behaviour without changing the language that the parser accepts, and without introducing new conflicts elsewhere. Quite often, seemingly trivial changes "rippled" through large parts of the grammar, causing numerous modifications. Furthermore, errors in the ISO grammar had to be identified and worked around.

Several techniques were used for correcting aspects of the grammar that the parser generator cannot deal with. They include:

Resolving shift/reduce conflicts by assigning precedence values to tokens or rules. Many of the changes of this kind are carried out according to the precedence rules in [ISO93, 9.8].

The given precedence rules cause problems in some cases. Most notably, the character '`*`' is given different precedence depending on whether it is used in type composition ("int $\times$ bool") or in expressions ("$56 \cdot 3$").

In other cases, precedence is assigned to tokens which are not treated specially in the ISO grammar, to resolve shift/reduce conflicts for which the desired action of the parser is always to shift, or always to reduce (i.e. there is no ambiguity). For example, the character '`;`' is given highest precedence in order to allow graceful recovery from parse errors at the next occurrence of '`;`'.

"Inlining" of the definitions of sub-goals. To delay the reduction of goals, this method takes advantage of the stack used by bottom-up parsers. For example, the following grammar is not accepted by a parser generator:

input = *x* | *y* ;
x = *a* "+" "x" ;
y = *b* "+" "y" ;
a = "?" | "a" ;
b = "?" | "b" ;

The parser cannot deal with inputs like "?+y" because after the "?" has been read, it needs to be reduced to either *a* or *b* before being pushed onto the stack. (It is not possible to push the "?" itself as the reduce action would then later on have to be performed on something that is not at the top of the stack.) However, at this point the lookahead token is "+", so it is not yet clear whether the non-terminal being parsed is *x* or *y*. Inlining of the definitions of *a* and *b* solves the problem:

input = *x* | *y* ;
x = "?" "+" "x" | "a" "+" "x" ;
y = "?" "+" "y" | "b" "+" "y" ;

Note that the single characters in the simple example above are often arbitrary non-terminals in the VDM-SL grammar. Inlining is used extensively in `parse.y`, even though it has the disadvantage that the complexity of the grammar increases noticeably.

Simulating additional lookahead tokens. Sometimes, inlining of sub-goals is not a solution, as in the case of the $\in$ operator, written as the two words "in set" in VDM-SL. The grammar rules involved are as follows (only the rule alternatives relevant to the problem are given):

expression = **def** *patternbind-expr-list* **in** *expression*
| name
| *expression* **in set** *expression* ;

patternbind-expr-list = *pattern-bind* "=" *expression* ;

pattern-bind = *pattern*
| *bind* ;

pattern = *name*
| *pattern* **union** *pattern* ;

bind = *pattern* **in set** *expression* ;

The problem occurs because during the parsing of an *expression*, it cannot be determined whether the end of *patternbind-expr-list* has been reached, as any "in" following could be either the "in" after the *patternbind-expr-list* or the start of "in set" followed by another expression. All attempts to inline some non-terminals fail because different problems arise from the changes.

The solution is simulating a further lookahead token to allow the parser to distinguish between "in" and "in set". The parser itself cannot be altered to do this, but a common trick used in such a situation is to make the scanner return a special terminal token. In this case, the two words "in set" are not returned as two tokens IN, SET. Instead, they are combined into just one token IN_SET.

Due to the requirement that IN_SET is returned even when "in" and "set" are separated by any sequence of whitespace and/or comments, it is more appropriate not to change the scanner itself, but to insert a new "layer" between the scanner and the parser.

This behaviour is implemented in the file `lexqueue.cc`: All tokens are passed straight on to the parser, with one exception: If the token is IN, the scanner is called once more to see whether the next token is SET. If it is, IN_SET is returned to the parser, and if it is not, IN is returned and the token following it is buffered and returned the next time the parser requests a token.

"Trial and error". While this is not the most sophisticated way of proceeding, trying out a few alternative ways of expressing the same language construct is often a good alternative to exploring the problem in detail, which can take some time.

The following ambiguities and errors in the ISO grammar were the most severe ones – considerable effort was necessary to identify/work around them:

- The VDM-SL representation of the VDM expression "X-set" appears to be "set of X", even though this is not mentioned in [ISO93, 10].
- It is not clear what the difference between the VDM-SL map operators merge and munion is.
- There is reason to believe that the lambda operator is a member of the family of *constructors* (see [ISO93, 9.8]), but this is not explicitly mentioned – the operator needs to be assigned a precedence value according to what family of operators it is in.
- The *identifier* non-terminal does not seem to stand for the same thing in all parts of the grammar: In some cases, it only denotes those identifiers beginning with "is_" or those beginning with "mk_". The absence of separate non-terminals for these cases is the cause of numerous ambiguities.

  This particular error in the ISO grammar needed more than 10 hours to identify. The distinction between the different identifier types is implemented with special code in the scanner, which checks for a is_ or mk_ prefix.

In addition to the changes described above, which are motivated by the need to transform the grammar into a grammar that the parser generator can process, there are also changes to support later stages of the program: The non-terminals *openScope*, *closeScope*, *openComprehensionScope*, *beginPreCond* and *endPreCond* are all empty, i.e. their rules are of the form

openScope =

Hence, inserting them into any rules does not alter the language that the parser recognizes. However, the action associated with each of the rules is used to influence how subsequent input is parsed.

Finally, as an example of the grammar transformation process, here is the rule for *let-be-expression* the way it appears in the ISO grammar and in `parse.y`. The rules in the ISO grammar are as follows:

expression = *let-be-expression* | ... ;
let-be-expression = **let** *bind* **be st** *expression* **in** *expression* | ... ;

bind = *set-bind* | *type-bind* ;
set-bind = *pattern* **in set** *expression* ;

Inlining has been applied several times to these rules, as it was necessary to isolate the problematic "in set" case. Since the one "physical" rule in `parse.y` represents more than one "logical" ISO grammar rule, the code associated with it needs to construct more than one parse tree object. It does this with calls to the macro MK, whose definition is also given. The dl() functions simply delete all of their arguments before throwing a bad_alloc() exception. The rule is assigned the precedence of the family of constructors with a %prec CTOR declaration.

```
#define MK(_dest, _ToCreate, _dels) \
  if ((_dest = new(nothrow) _ToCreate) == 0) { dl _dels; }

expression:
    LET openScope pattern IN_SET expression BE ST expression IN expression
    closeScope %prec CTOR {
        SetBind* sb; MK(sb, SetBind($3, $5), ($3, $5, $8, $10));
        MK($$, LetBeExpr(sb, $8, $10, $1.pos), (sb, $8, $10)); } ;
```

4.2.2 Representation of the Parse Tree

As described in section 3.2, the parse tree is represented by nodes of objects whose classes are derived from Tree::N. The class declaration is as follows – Dassert is described in section 4.4.3.

```
class N {
public:
  /* no ctor of a class derived from N should throw any exception other
     than bad_alloc. dtors should not throw any exceptions. */
  N() throw(std::bad_alloc) { }
  inline virtual ~N() throw() = 0;
  virtual ostream& put(ostream& s) const = 0;
private:
  N(N&) { Dassert(false); abort(); } // need deep copy
  void operator=(const N&) { Dassert(false); abort(); } // need deep copy
};
N::~N() throw() { }

inline ostream& operator<<(ostream& s, const N& t) { return t.put(s); }
```

The classes are declared in the file `tree.h`. For those classes in the hierarchy whose constructors are not inline, the constructors are defined in `mktree.cc`, whereas non-inline destructors are located in `tree.cc`. The definitions of the put() methods (used for printing the tree to the screen) are defined in the file `printtree.cc`.

The class hierarchy is quite "flat"; most classes derive either directly from N or from an abstract class that is derived from it.

Tree::TypeDefinition is an example of a class that derives directly from N. It also illustrates the close relation between the class layout and its corresponding non-terminal rule in the parser. Furthermore, it serves to show how null pointers are used if optional sub-goals of the rule are not present for a particular instance. Here are the grammar rules for *type-definition* from `parse.y`, followed by the class declaration for TypeDefinition from `tree.h`:

```
type_definition:
     name '=' type { MK($$, TypeDefinition($1, $3), ($3)); }
   | name '=' type invariant {
            MK($$, TypeDefinition($1, $3, $4), ($3, $4)); }
   | name DBL_COLON field_list {
            MK($$, TypeDefinition($1, $3), ($3)); }
   | name DBL_COLON field_list invariant {
            MK($$, TypeDefinition($1, $3, $4), ($3, $4)); }
   ;

class TypeDefinition : public N {
public:
   inline TypeDefinition(IdToken* id, N* tf, Invariant* i = 0);
   inline TypeDefinition(TypeDefinition& t);
   inline virtual ~TypeDefinition();
   virtual ostream& put(ostream& s) const;
   const IdToken& identifier() const { return *identifierVal; }
   IdToken& identifier() { return *identifierVal; }
   inline const Type& type() const;
   inline const FieldList& fieldList() const;
   bool hasInvariant() const { return invariantVal != 0; }
   const Invariant& invariant() const {
      Assert(hasInvariant()); return *invariantVal;
   }
   Invariant& invariant() {
      Assert(hasInvariant()); return *invariantVal;
```

```
  }
private:
  IdToken* identifierVal;
  N* typeOrField;
  Invariant* invariantVal;
};
```

In contrast to TypeDefinition, it is convenient to have an abstract class DefinitionList for the classes TypeDefinitionList, ValueDefinitionList, OperationDefinitionList and FunctionDefinitionList. The rule for *type-definition-list* and the class declaration for TypeDefinitionList below also show how list-like non-terminals are implemented using vector:

```
type_definition_list:
    TYPES type_definition { MK($$, TypeDefinitionList($2), ($2)); }
  | type_definition_list ';' type_definition {
          try { $1->push_back($3); }
          catch (...) { delete $1; delete $3; throw; } }
  | type_definition_list ';' error { }
  | type_definition_list error { }
  ;

class DefinitionList : public Definition {
public:
  DefinitionList() { }
  inline virtual ~DefinitionList() = 0;
  virtual ostream& put(ostream& s) const = 0;
};
DefinitionList::~DefinitionList() { }

class TypeDefinitionList
    : public DefinitionList, public vector<TypeDefinition*> {
public:
  explicit TypeDefinitionList(TypeDefinition* td)
      : vector<TypeDefinition*>(1, td) { Dassert(td != 0); }
  virtual ~TypeDefinitionList();
  void push_back(TypeDefinition* td) {
    Dassert(td != 0); vector<TypeDefinition*>::push_back(td);
  }
  virtual ostream& put(ostream& s) const;
};
```

4.2.3 Scope Handling

The term *name scope* is used to describe a feature found in many computer languages, including VDM: Identifiers are only valid in a certain section of the input file and using them elsewhere is not allowed, typically because it would not make sense to use them. Some examples of this behaviour for VDM are:

- The identifiers for function/operation arguments are only valid within the definition of that function or operation.
- In constructs like "let x = 1 in some_expression", the identifiers that are introduced (in this case, 'x') are only valid until the end of the expression has been reached.

Furthermore, a local declaration may shadow a declaration of the same name in an enclosing scope. That shadowed name is inaccessible, but becomes available again after the local scope has been closed.

It was originally hoped that no support for scopes would need to be implemented, but the following problem made this necessary: In operation pre-conditions, the external state variable may be referred to either with or without a trailing '~', i.e. either as the old or the new value, without any semantic difference. Thus, the following two operation definitions are equivalent:

```
opname() r: bool                opname() r: bool
  ext   wr x: int                 ext   wr x: int
  pre   x = 0                     pre   x~ = 0
  post  r <=> true                post  r <=> true
```

When the pre- and post-condition are combined using a logical and during the partitioning pass, all references to the state in the pre-condition must obviously be to the *old* state, so for both cases, the expression "x~ = 0 and r <=> true" must be generated.

In order to decide correctly when to turn a 'x' into a 'x~', it must be known whether the expression currently being parsed is part of a pre-conditition (this is achieved by modifying a flag with the *beginPreCond* and *endPreCond* non-terminals), and whether 'x~' is defined in the local scope.

The implementation of scope support also includes code to deal with the two error conditions that a name is declared twice in a scope, or that a name is used even though it has not been declared – see page 48 in section 4.6.2 for an example of the error message.

The interface and code for scope support are located in the files `scope.h` and `scope.cc`. The program creates one VarDef instance for each variable and stores a pointer to it in the parse tree. This VarDef object uniquely identifies the variable, function etc. that is being referred to, even if the spelling of two different names is identical – if 'x' is both a function's local variable and a function name, there will be two VarDef objects for 'x'.

The data structures used for scope handling are probably the most complicated data structures used in the program. They were designed with the following goals:

- Lookup of a VarDef object given an identifier spelling should be fast, i.e. the scanner's hash table should be used.
- There should be no overhead when looking for the most locally defined instance of a name. For example, if 10 name scopes are currently open and a name is requested that is only present in the outermost, top-level scope (or *global scope*), the program should not have to search linearly through the other 9 scopes before arriving at the definition in the top-level scope.
- The semantics described above should be correctly modelled, including shadowing of name definitions.
- The following common operations should be cheap: Creating a new local scope that is "active" (i.e. accessible at the moment); closing an active scope (by moving it to a list of "old" scopes); inserting a new VarDef into the local scope or one of the scopes surrounding it.

The implementation ensures these properties by using a grid-like structure of objects: In one direction, VarDef objects whose names have the same spelling are accessed through the hash table and are singly linked in the reverse order in which the scopes were opened, i.e. a pointer from one of the IdTokens in the hash table points to the most locally defined variable with that spelling[7]. In the other direction, VarDef objects that are declared in the same scope are connected as a singly-linked list to support operations on whole scopes, e.g. closing them. The start of each such list is stored in an ordered list activeScopes for those scopes that are currently accessible, or in an unordered collection oldScopes for those scopes that are no longer accessible. Finally, each VarDef object contains a pointer to the IdToken with its spelling.

[7] Thus, looking up a variable definition given its spelling is close to O(1) with a good hash table implementation.

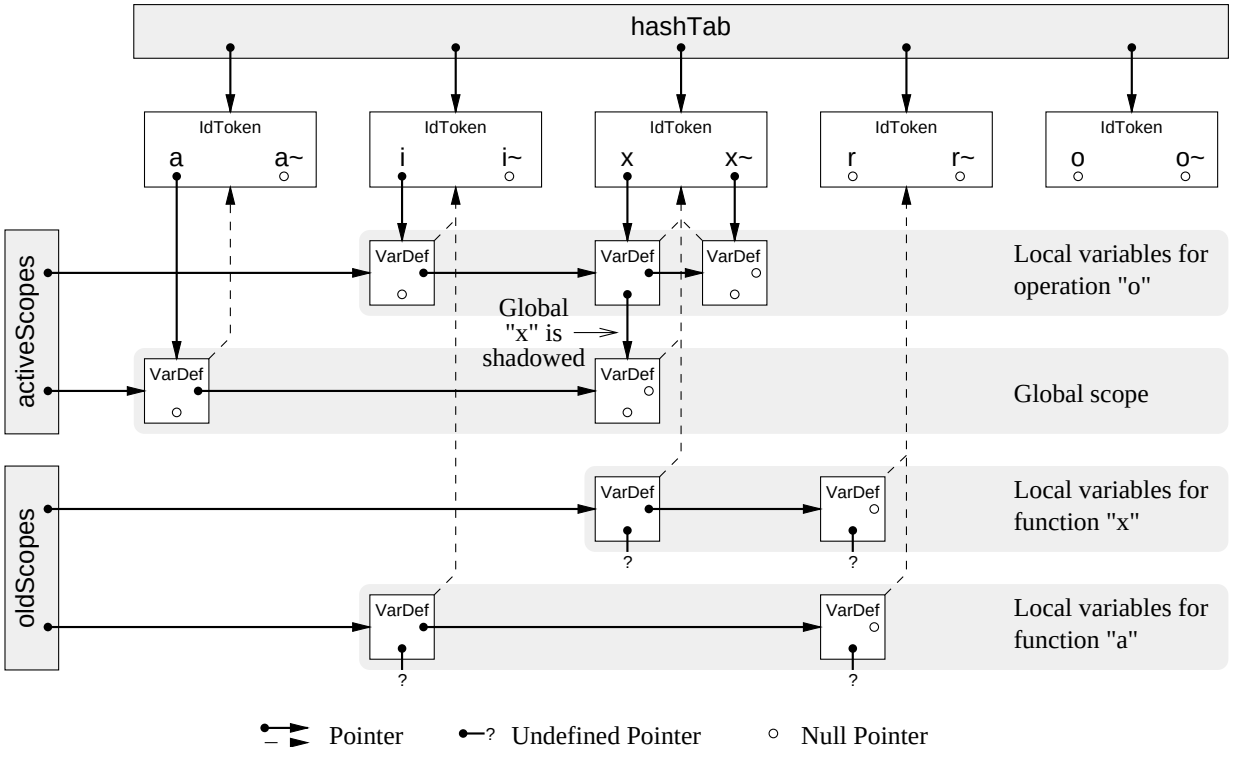

Figure 1: Data structures created for the example code, during the parsing of “o”

vdmpart’s scope handling is illustrated below with an example. Figure 1 shows the current state of the data structure at the moment that the post-condition of operation o in the specification below is parsed. Notice that no VarDef object for o itself has yet been inserted at this point – this only happens *after* the whole operation definition has been parsed. This behaviour means that recursive calls are not possible, but the rest of the program does not support recursive invocation anyway.

```
functions
  a(i: int) r: int
  pre   i = 0
  post  r = i ;

  x(x: int) r: bool
  pre   x <> 5
  post  r <=> (x > 1)

operations
  o() i: int
  ext   wr x: int
  post  i = 5          -- state is examined here, before local scope is closed
```

4.3 Generation and Output of Test Data

The central function of vdmpart, which is passed a partition tree and generates a predicate expression for each possible permutation of the sub-partitions, is only 17 lines long. Its definition is located in the file `partition.cc`:

```
void createTestCases(ostream& s, const Part::Partition* part) {
  BigInt comb(part->combinations());
  s << "# " << comb << " test cases\n";
  if (comb == 0U) return;
  BigInt limit(comb);
  size_t bits = limit.roundUp();
  BigInt count;
  SmartPtr<Tree::Expr> testCase;
  do {
    if (count < comb) {
      testCase = part->partition(count);
      s << "# " << count << '\n' << *testCase << '\n';
    }
  } while (count.mirrorInc(bits) == success);

  return;
}
```

createTestCases() directly and indirectly makes use of all other components that are connected with the final stage of the program. Their implementation is described in the following sections.

4.3.1 Functions For Partitioning

The four virtual methods partTrue(), partFalse(), partDef() and partUndef() are introduced as methods of the abstract base class Tree::Expr, and are consequently inherited by all parse tree nodes that represent expressions.

For expression nodes that the program supports (e.g. integer division), the class declarations provide their own versions of these methods. For the other, unsupported expression classes, the default methods, e.g. Tree::Expr::partDef(), are called; they cause an error message to be printed and the partitioning process to be aborted. Again, an example of the error message can be found in section 4.6.2, on page 49.

The declaration of the class Expr is given below, together with that of its child class DivExpr. Both are taken from `tree.h`:

```
class Expr : public N, public SmartPtrBase {
public:
  explicit Expr(const TextPos& pos) : posVal(pos) { }
  virtual ostream& put(ostream& s) const = 0;
  const TextPos& pos() const { return posVal; } // NB non-virtual
  virtual Part::Partition* partDef();
  virtual Part::Partition* partUndef();
  virtual Part::Partition* partTrue();
  virtual Part::Partition* partFalse();
private:
  const TextPos posVal;
};

class DivExpr : public BinaryExpr {
public:
  DivExpr(Expr* l, Expr* r, const TextPos& pos) : BinaryExpr(l, r, pos) { }
  virtual ostream& put(ostream& s) const;
  virtual Part::Partition* partDef();
  virtual Part::Partition* partUndef();
  virtual Part::Partition* partTrue();
  virtual Part::Partition* partFalse();
};
```

The definitions of DivExpr's methods are located in the file `partvirt.cc`. Since partTrue() and partFalse() are only called for expressions that return boolean results, their being called for an

integer division is a clear indication that there is a semantic error in the input (e.g. the expression reads "(x div y) and true"), so an error message is printed with a call to shouldBeBool():

```
Part::Partition* Tree::DivExpr::partDef() {
  return Part::Def::integerDiv(&first(), &second());
}
Part::Partition* Tree::DivExpr::partUndef() {
  return Part::Undef::integerDiv(&first(), &second());
}
Part::Partition* Tree::DivExpr::partTrue() {
  return shouldBeBool(this);
}
Part::Partition* Tree::DivExpr::partFalse() {
  return shouldBeBool(this);
}
```

As specified in section 5, DivExpr::part$X$() only consists of a call to Part::X::integerDiv(). The actual partition generation is then performed in that function. The code below is the implementation for Undef::integerDiv() from `partition.cc`. It is preceded by the corresponding partitioning rule for partUndef(e_1 div e_2), as given on page 12:

$$\text{partUndef}(e_1 \text{ div } e_2) = \left\{ \begin{array}{c} \text{partDef}(e_1) \\ \text{partUndef}(e_1) \end{array} \right\} \bigtimes \left\{ \begin{array}{c} \text{partTrue}(e_2 \mathrel{<>} 0) \\ \left\{ \begin{array}{c} \text{partTrue}(e_2 = 0) \\ \text{partUndef}(e_2) \end{array} \right\} \end{array} \right\}$$

```
Part::Partition* Part::Undef::integerDiv(Expr* a, Expr* b) {
  SmartPtr<PartSet> pset(new PartSet());
  pset->reserve(3);
  Partition* defA    = Def::lookup(a);
  Partition* undefA  = Undef::lookup(a);
  Tree::ExprPtr zero(new Tree::IntegerExpr(0, b->pos()));
  Tree::ExprPtr bEqual0(new Tree::EqualExpr(b, zero.get(), b->pos()));
  Partition* defB    = False::lookup(bEqual0.get());
  SmartPtr<PartSet> undefB(new PartSet());
  undefB->reserve(2);
  undefB->push_back(True::lookup(bEqual0.get()));
  undefB->push_back(Undef::lookup(b));
  pset->push_back(new FullComb(undefA, defB));
  pset->push_back(new FullComb(defA, undefB.get()));
  pset->push_back(new FullComb(undefA, undefB.get()));
  return pset.release();
}
```

The source code models the specified behaviour with the following steps (e_1 and e_2 correspond to a and b):

- Create a PartSet (pointed to by pset) with three entries, for enumerating the three possibilities indicated by the bipartite graph operator.
- Create partitions for the left hand argument of the bipartite graph operator: "a defined/undefined" is represented by (defA/undefA).
- The division partitioning rule is special in that it introduces a new expression which is not already a part of the parse tree. Hence, the partition for the expression "b = 0" can only be constructed after that expression itself has been created. Do this by allocating an Expr for the constant "0" (zero in the code) and using it to create bEquals0.
- Now create partitions for the right hand argument of the bipartite graph operator. bEquals0 can be used twice because partTrue(e_2 <> 0) is equivalent to partFalse(e_2 = 0). defB is the partition which represents "b is not zero".
  Create another PartSet for the two alternatives "b is zero" and "b is undefined" which make up undefB.
- Finally, enumerate the three possibilities of the bipartite graph operator by connecting with FullComb operations the partitions obtained during the previous steps, and adding them to pset with push_back().

The calls to the Part::X::lookup(x) functions implement the dynamic programming: They return the result of the call x->part$X$(), but they also store this result, so the partioning call is only made once for each expression; on subsequent calls, the value is returned immediately after a table lookup.

Throughout the program, attention was paid to the possibility that a function call may raise an exception, and the code was made exception-safe. In the example of Undef::integerDiv() above, this is reflected in the fact that reserve() is called for the vector part of the PartSet, so that push_back cannot fail – if it failed, the new FullComb that was about to be added to the vector would be left behind unreferenced. Furthermore, the use of SmartPtr ensures that objects are automatically deleted once the function returns, except when release() has been called for the SmartPtr.

One may argue that this level of exception safety is overzealous since the program will be terminated with "out of memory" (or similar) anyway, and indeed it is not necessary for vdmpart,

but forgetting about the possibility of exceptions can lead to serious and difficult-to-find bugs in other types of programs, so it is a good idea *always* to take it into consideration.

4.3.2 Representation of the Partition Tree

The implementation of the partition tree representation is similar to that of the parse tree – the largest difference is that there are far fewer classes that derive from the abstract base class Part::Partition. As specified in section 5, they are called FullComb, PartSet, ExprSet and ConstTrue/ConstFalse.

Another important difference between the parse tree and the partition tree is that the latter is not actually a tree, but just a directed, a-cyclic graph in many cases, due to the fact that the same partition tree object may be pointed to by two or more other objects. This must be permitted in order for the dynamic programming technique to be implemented, which, as mentioned on page 12, is used to avoid repeated evaluation of the same expression during the generation of the partition tree.

This sharing of objects introduces a problem: The destructor of each Partition class is expected to delete the objects it points to – however, if more than one reference to an object exists, simply using delete results in attempts to delete the object more than once, with disastrous consequences. On the other hand, not deleting the object at all is a memory leak.

The problem is solved with the introduction of a reference count that is stored in all Partition objects and updated whenever a new reference to the object is created or destroyed. The SmartPtr template used for this is described in detail in section 4.4.1 – the only consequence for the implementation of Partition is that the class must be derived from SmartPtrBase.

The declaration of the Partition base class below introduces virtual methods that are implemented for all the classes deriving from it, including combinations() to return the number of predicates that a partition tree node can "produce", and partition() to create and return one of these predicates. It is taken from the file `partition.h`.

```
class Part::Partition : public SmartPtrBase {
public:
  Partition() { }
  virtual inline ~Partition() throw() = 0;
  virtual bool isConstTrue() const throw() = 0;
  virtual bool isConstFalse() const throw() = 0;
  // nr of possible permutations for this node (and the nodes it contains)
  virtual const BigInt& combinations() const throw() = 0;
```

```
    /* create expr. corresponding to particular permutation of
       sub-partitions. arg must not be out of range. */
    virtual Tree::Expr* partition(const BigInt& x) const = 0;
    virtual ostream& put(ostream& s) const = 0;
  protected:
    static const BigInt zero;
    static const BigInt one;
  private:
    // not necessary ATM:
    explicit Partition(Partition&) : SmartPtrBase() {
      Dassert(false); abort(); // virtual needed?
    }
    void operator=(const Partition&) {
      Dassert(false); abort(); // ditto?
    }
};
Part::Partition::~Partition() throw() { }

inline ostream& operator<<(ostream& s, const Part::Partition& t) {
  return t.put(s);
}
```

The child classes of Partition are laid out according to the concept they represent. Whereas PartSet and ExprSet use vectors to store sets of sub-partitions/expressions, ConstTrue/ConstFalse are just dummy classes that do not add any data members.

FullComb, whose declaration is given below, contains pointers to the two sub-partitions to which the full combination operator is applied – more accurately, PartPtr (which is typedef'd to SmartPtr<Partition>) is used instead of Partition*. The isConstTrue/False methods are implemented in such a way that any occurrances of ConstTrue/ConstFalse objects are eliminated.

```
class Part::FullComb : public Partition {
public:
  FullComb(Partition* l, Partition* r);
  virtual ~FullComb() { }
  Partition& left() const { return *leftVal; }
  Partition& right() const { Assert(!rightVal.isNull()); return *rightVal; }
  virtual bool isConstTrue() const {
```

```
    if (rightVal.isNull()) return leftVal->isConstTrue(); else return false;
  }
  virtual bool isConstFalse() const {
    if (rightVal.isNull()) return leftVal->isConstFalse(); else return false;
  }
  virtual const BigInt& combinations() const { return combinationsVal; }
  virtual Tree::Expr* partition(const BigInt& x) const;
  virtual ostream& put(ostream& s) const;
private:
  PartPtr leftVal;
  PartPtr rightVal;
  BigInt combinationsVal;
};
```

4.3.3 Building the Test Case Predicates

As soon as the partition tree has been built, the individual test case predicates can be requested with calls to partition(x) for the root object of the partition tree, where x is a BigInt between zero and the value returned by combinations() for that node.

The implementation of partition() creates the predicate by calculating which part of the partition tree is relevant for the BigInt argument given, recursively calling partition() for the appropriate sub-partition(s) (but with a different argument) and, if more than one sub-partition is involved, combining the expressions into one.

As shown in section 3.3.3, the number of combinations $c(p)$ for a whole PartSet is the sum of the number of combinations for all the n sub-partitions it contains; $\sum_{i=1}^{n} c(p_i)$. The implementation of partition(x) for this node type must reverse the sum operation in the sense that it must find the smallest m so that $\sum_{i=1}^{m-1} c(p_i) \leq x$. Next, it must call partition() for the mth sub-partition with an argument of $x - \sum_{i=1}^{m-1} c(p_i)$.

For example, if a PartSet contains three subpartitions with 300, 200 and 100 combinations, its partition() method can be called with values from 0 to 599. If it is called with a value of 505, the third sub-partition is selected (since $300+200 \leq 505$) and the third sub-partition's partition() method is called with an argument of 5.

The code for PartSet::partition() performs a binary search to find the right sub-partition. A simple linear search would probably even have been more efficient in this case because at present, no PartSet ever contains more than three sub-partitions, but binary search was favoured nevertheless in anticipation of the possible improvements to the algorithm described on page 17. If they are ever carried out, PartSets will sometimes contain hundreds or thousands of sub-partitions.

In the FullComb class's version of partition(), the expressions returned by recursive calls for the left-hand and right-hand operands to the full combination operator are combined with a logical and. The value returned by combinations() for a FullComb object is the product of the number of combinations for the two sub-partitions, so the implementation only needs to perform an integer division/remainder calculation to determine the argument values for the two recursive calls to partition(). This code excerpt is taken from `partvirt.cc`:

```
Tree::Expr* Part::FullComb::partition(const BigInt& x) const {
  BigInt val, rem;
  if (rightVal.isNull()) return left().partition(x);
  x.divrem(left().combinations(), val, rem);
  SmartPtr<Tree::Expr> l(left().partition(rem));
  SmartPtr<Tree::Expr> r(right().partition(val));
  return new Tree::AndExpr(l.get(), r.get(), TextPos::none);
}
```

4.3.4 The BigInt Integer Abstraction

The BigInt concrete type is implemented in `bigint.h` and `bigint.cc` according to the description in section 3.3.3. Since the class declaration is straightforward, it is not reproduced here.

However, one part of the code that deserves to be mentioned is how mirrorInc() has been carried out: Rather than mirroring the integer bit by bit, adding one and then mirroring it again, mirrorInc() performs the increment operation directly on the mirrored representation. This is the function definition from `bigint.cc`:

```
bool BigInt::mirrorInc(size_t n) { // increase by 1
  while (n > 0) {
    ValueType toggle = 1 << --n;
    if (((val ^= toggle) & toggle) != 0) return success;
  }
  return failure;
}
```

4.4 Further Components of the Implementation

This section of the dissertation describes some important components of the implementation that have the common property that they cannot be associated with any particular stage of the test data

generation. Instead, they are "standard components" which are general-purpose – as such, they can be re-used for other programs.

4.4.1 The SmartPtr Template

"Smart pointers" are a useful concept in C++ to improve the low-level memory allocation scheme of the language. They allow the programmer to express which data structures "own" other data structures and which only reference them. Furthermore, just like the library template auto_ptr, they make it easy to improve exception safety. For further discussion of these issues, see [Stroustrup97, 14.4].

The idea behind the so-called "resource acquisition is initialisation" technique supported by smart pointers is to provide a class that only contains a pointer, and for which operators have been overloaded in such a way that its usage is very similar to that of a pointer. An important feature of a smart pointer class is that when its destructor is called, it may delete the object it is pointing to. By making smart pointers local objects of a function, this feature can be used to have objects deleted automatically when the local scope is destroyed – regardless of whether this happens because the function returns or because an exception has been raised.

A SmartPtr<X> behaves much like a regular X*, with the difference that the X object always contains a count which represents the number of smart pointers pointing to it. When an X object is first created, the count is zero. Subsequently, it is updated whenever smart pointers to it are created, assigned or destroyed. If the count ever reaches zero during a call to ~SmartPtr<X>, the X object is deleted. X must have been derived from SmartPtrBase if it is to be used with smart pointers.

The SmartPtr template and a number of further functions are defined in `smartptr.h`. They provide the following interface:

- Initialisation of SmartPtr<X> objects with the null pointer, with a pointer to X or to a class derived from X, or with another SmartPtr pointing to X or to a class derived from X.
- Assignment to SmartPtr of a pointer or smart pointer, which may point to the same class or a class derived from it. Assigning null requires an explicit cast, e.g. "ptr = (X*)0;".
- Dereferencing (prefix '*') and indirection (infix '->') work just like for ordinary pointers.
- Likewise, the comparison operators '<', '>', '<=' and '>=' result in comparisons of the pointers contained within the SmartPtr objects. For '==' and '!=', one of the objects can even be a regular pointer instead of a smart pointer.

- In contrast, the meaning of the "address of" operator (prefix '&') has *not* been changed – it returns a pointer to the SmartPtr object. The method get() returns the address of the object that the smart pointer references.

- The release() method is equivalent to assigning null to the SmartPtr, except that the object being pointed to is never deleted, not even if that SmartPtr was the last one referencing it.

- swap() (available both as a method taking one argument and a global function taking two arguments) is an efficient way of swapping the contents of two SmartPtr objects.

- For convenience, the method isNull() is provided to make testing for null possible without an explicit cast.

- The global function makeSmartPtr(x) creates a SmartPtr to the class of x without the need for the caller to specify the class name. This is convenient for cases when only a normal pointer to an object is available, but a function needs to be called which takes a SmartPtr argument; no local SmartPtr variables need to be created.

- The global function deleteSmart(x), which takes a normal pointer as an argument, performs "delete x;" if the reference count for the object pointed to by x is zero.

- The global function releaseSmart(x) is also given a normal pointer as its argument. Its only effect is that the reference count for the object pointed to by x is decreased by one. It must be used with care because it can easily result in an object being deleted too early, i.e. while other SmartPtrs to it still exist.

- If instances of classes that derive from SmartPtrBase are not created on the heap, their reference count defaults to zero. This is not desirable because it can result in attempts by ~SmartPtr to perform a delete operation on them. The memberless class SmartPtr_lockStatic is designed to overcome this problem: If a SmartPtr_lockStatic is defined immediately after the definition of the variable, and is initialised with a reference to that variable, it will increase its reference count by one. For example, a global variable can be "locked" against deletion like this:

```
SmartPtr<X> x;
SmartPtr_lockStatic lock(x);
```

Note that the SmartPtr_lockStatic object must be defined *after* the SmartPtr, and must be defined in the same compilation unit – otherwise, the order of initialisation is not guaranteed.

The definition of the SmartPtr template probably constitutes the most advanced usage of C++ in the program. For this reason, a cut-down version of the code (without the debugging support) is included in this dissertation despite its length. It is reproduced in appendix D.1.

4.4.2 Error Handling

All functions related to the reporting of errors are located in the files `error.h` and `error.cc`. There are three types of error messages: Genuine errors, warnings, and "not implemented" errors. A count of the number of errors and warnings that occurred so far is maintained in the variables Error::errorCount and Error::warningCount and can be used in other parts of the program, e.g. to abort execution before the test data generation stage in case there were any errors during parsing.

The three functions error(), warning() and unimplemented() print the appropriate type of message. There are four variants of each of the functions, since the text to be printed in the message can be supplied either as a const string object or as const char*, and additionally, the function can optionally be given a TextPos argument containing line number information. For error(), the function prototypes are:

```
void error(const string& s, const TextPos& t);
void error(const char* s, const TextPos& t);
void error(const string& s);
void error(const char* s);
```

Calls to these functions are made from many different parts of the program. Section 4.6.2 lists numerous examples for the output they generate.

4.4.3 Debugging Aids

Experience has shown that it makes sense to leave some debugging code enabled even in "release" versions of a program. In particular, assertions (i.e. checking for the violation of invariants) are often very useful to track down a bug that a user has encountered in the program: Instead of just crashing, the program will print a message that can help to identify the source of the problem.

On the other hand, the release version should not contain *all* assertions, either. Often, the tests are very simple and only serve the purpose of catching programming errors during the implemen-

tation stage. They may also be so numerous that they considerably increase the executable size or – if they are located in inner loops or inline functions – the program's performance.

The policy that was adopted for vdmpart was only to compile in those assertions in all cases that check for violations of the interface between individual components of the program. Additional checks within each component are only enabled if the program is compiled with debugging support, using the `-DDEBUG` compiler switch (which is passed on by make if the command "`make X=-DDEBUG`" is used).

As the standard ANSI C library assert() facility only allows to either compile in all assertions or none, a slightly enhanced version is provided in the file `debug.h`. It defines two macros Assert() and Dassert(). Both of them check whether a condition is fulfilled and print an error message to stderr if it isn't. However, whereas Assert() checks are always compiled into the program, the Dassert() macro only produces code if debugging has been enabled with `-DDEBUG`. Unlike the standard library assert(), Assert() and Dassert() do not call abort() if the assertion fails, but let program execution continue.

4.5 Tests of the Program Components

The various components that make up vdmpart have been identified and separated from each other in previous sections. Before they were integrated into one program, each component was tested separately.

Scanner Because the scanner directly processes the input, testing it was easily achieved by confronting it with a selection of inputs consisting of valid and invalid tokens. Debugging code (triggered by a command line switch) prints out the value of the token that is output, and, if necessary, any additional information such as the contents of a string constant.

Parser The interface between scanner and parser is trivial and the scanner/parser generator tools are designed to work with each other, so the concept of separate testing was violated in this case; the parser was given its input by the scanner during testing. The parser generation tool includes an option to generate debugging code which prints out the state of the stack, rules applied and other information. Together with the `y.output` file (created by the parser generator) that describes the generated states, it proved an invaluable tool for identifying problems with the grammar. This debugging information can be switched on with the `--debug-parse` switch.

The parser component also includes the parse tree. In order to determine whether the correct tree was being generated, the put() methods were implemented for all tree node objects.

Triggered by the `--debug-printtree` command line switch, they can be used to print out a textual representation of the parse tree. Inputs for *all* tree node types were written and the parser was presented with them – they now form part of the automatic test suite described in section 4.6.1.

Partitioning The turning of expressions into partition trees was tested with a small driver program which builds an expression by constructing a tree of Expr objects, and invoking one of the part...() methods on the top-level object. Analogous to the parse tree, put() was implemented for all partition tree classes. At this point, it became obvious that the ability to visualise the partition was not only useful for testing, but also for the users of the program, so output of partitions was made an "official feature". The `--partition` switch was added to enable it.

Even though the creation of equivalence classes has previously been described as belonging into a separate stage of the program, the implementation of the partition() method is dependent on the data structures used in the partition tree to such a degree that it makes sense to test the two together. Thus, testing was conducted by calling partition() for partition trees generated by the partitioning functions. The partition tree was first output using the `--partition` switch, and the displayed test cases were then verified manually.

Further Components The other, minor components of the system were not always tested separately, either because they are very simple (as in the case of the error reporting and debugging support) or because they were adapted to the changing needs of other components many times during program development – this happened with the SmartPtr code. While initial, separate testing *was* performed, this did not happen again each time the interface was changed.

4.6 Tests of the Final Program

4.6.1 Portability

In order to determine how portable vdmpart is, it was attempted to compile the program in a variety of environments. In general, there were few or no problems doing this. However, it became clear that the largest obstacle to compiling is that the C++ compiler used may not be recent enough to support the programming constructs and library facilities used by the program.

Linux Linux 2.2.13 running on the x86 architecture was the original development platform, consequently there were no problems during compilation. GCC 2.95 had to be installed to replace a slightly older version supplied with the original OS distribution.

Solaris The program was compiled without problems on two different Solaris systems:

On a x86 Pentium machine running Solaris 5.5, compilation was successful after GNU make and GCC 2.95 had been compiled and installed.

On a Sun Ultra-60 (two UltraSparc II processors, running Solaris 5.7), no additional work was necessary since GCC 2.95 had already been installed on the system earlier; compilation was immediately successful.

HP-UX An attempt to compile the program on a HP 9000/720 running HP-UX B.10.20 was abandoned because it turned out that too many components of the system needed upgrading in order to run GCC 2.95.

CygWin (Windows) CygWin by Cygnus Solutions provides a Unix-like environment under Windows 98 and Windows NT. To make vdmpart compile properly, a very minor change had to be made to the getopt_long() call made by the program.

An addititional measure was taken to ensure that any problems with the compiled program are noticed: A small test suite is supplied in the `examples` subdirectory of the source distribution. The command "make check" causes the program to be run on a number of input files in this subdirectory and its output to be compared with that of a version of the program that is known to work. If there are any differences between the expected and the actual output, "make check" produces an error. For example, if there were a problem with the "err08" test, the output of "make check" might look like this:

```
examples> make check
gawk -f check.awk
Testing ‘division’...     OK
Testing ‘err01’...        OK
Testing ‘err02’...        OK
Testing ‘err03’...        OK
Testing ‘err04’...        OK
Testing ‘err05’...        OK
Testing ‘err06’...        OK
Testing ‘err07’...        OK
Testing ‘err08’...        FAILED!
Output was:
```

```
    err08.t.out - differ: char 255, line 8
Testing ‘err09’...        OK
Testing ‘err10’...        OK
Testing ‘huge’...         OK
Testing ‘relations’...    OK
Testing ‘tree01’...       OK
Testing ‘tree02’...       OK
Testing ‘tree03’...       OK
Testing ‘tree04’...       OK
Testing ‘tree05’...       OK
Testing ‘tree06’...       OK
Finished: 1 out of 19 tests failed (5%)
make: *** [check] Error 1
```

Obviously, in practice none of the tests should fail.

4.6.2 Robustness

The robustness tests of the program can be subdivided into two categories: Very large, but correct input, and input with syntactic and/or semantic errors. (An example of a small, correct input file together with the resultant output is given in appendix B.)

For the first category, the following input was given to vdmpart:

```
operations

  operation(a, b: int, c: bool, d, e, f, g: int, h, i: bool) r: int
  ext wr x: int
      rd y, z: bool
      wr s, t, u, v: int
  pre x~ + s + t = v * u * d
      or ((h => i) and (y and not z <=> false) or (not c))
  post (d div e >= f or a < b or b < a) => g <= f
      or x div g <= 0 and r = a div b div d
```

The input expressions may not seem to be *very* large, but as the number of generated test cases increases exponentially, a huge number of cases is output. vdmpart reported the generation of

129 537 408 test cases. Generating them and counting the number of characters that were output took 10 hours on a 266 MHz Pentium II. The size of the data that was output amounted to 38 GB.

The reaction of the program to the second category of input is best illustrated with a number of examples of the different error messages that it outputs, which are shown to the right of the input. Notice that a line number is output for most messages and that the program distinguishes between warnings, which are not considered fatal, and errors, which cause the test data generation to be aborted. In the examples below, only the message "`err05.t.vdm:8: Warning: Illegal use of ‘$’ in identifier`" is a warning.

```
1  state x of
2    a : int
3    b : rat
4    init x == 0
5    inv x == 0
6  end
```

```
> vdmpart err02.t.vdm -1
err02.t.vdm:6: Initialization must be
specified before invariant
```
(Lines 4 and 5 should be swapped)
```
err02.t.vdm: File does not declare any
implicit functions or operations
```

```
1  types
2    a = int
3      inv p == mk_(1)
```

```
> vdmpart err03.t.vdm -1
err03.t.vdm:3: ‘mk_(...)’  must be given at
least 2 expressions
```
(The requirement "at least two arguments" could have been encoded in the grammar, but it was more convenient to allow one or more arguments and to add an additional check.)

```
1  annotation
2  annotation
3  −− end annotation
4  "annotation" starts new
   annotation
5  '−−' annotation
6  :end
7    annotation
8  'end annotation
9  #end annotation
10 end annotation#
11 end annotation
12 end annotation−−
```

```
> vdmpart err04.t.vdm -1
err04.t.vdm:1: Unterminated annotation.
err04.t.vdm:2:     This line starts an
annotation
err04.t.vdm:4:     This line starts an
annotation
err04.t.vdm:4:     This line starts an
annotation
```

(Line 4 starts two nested annotations)

```
err04.t.vdm:7:     This line ends an
annotation
err04.t.vdm:8:     This line ends an
annotation
err04.t.vdm:9:     This line starts an
annotation
```

(Identifiers may contain “greek” characters – characters preceded by ‘#’ – this causes the ‘#’ in lines 9 and 10 to be considered part of the identifier.)

```
err04.t.vdm:11:     This line ends an
annotation
err04.t.vdm:12:     This line ends an
annotation
err04.t.vdm:13: parse error
```

```
1  types
2    a = int
3     inv p == 10E10 + '#i'
   + 'a' + id
4    + $types + i#d
5    + 66E
6    + '#j' -- no greek letter
7    + 'as' -- >1 character
8    + $id
9    + i#j
10   + "unterminated string
```

```
> vdmpart err05.t.vdm
err05.t.vdm: Sorry, not implemented:
Exponents for number constants.
err05.t.vdm:3: ‘id’ undeclared
err05.t.vdm:4: ‘$types’ undeclared
err05.t.vdm:4: ‘i#d’ undeclared
err05.t.vdm:5: Illegal numeric constant ‘66E’
err05.t.vdm:6: Illegal character constant
```

(Whereas “#i” stands for ι (iota), “#j” does not correspond to any greek character, so it is not allowed.)

```
err05.t.vdm:7: Illegal character constant
err05.t.vdm:7: Illegal character constant
err05.t.vdm:7: ‘s’ undeclared
err05.t.vdm:8: Warning: Illegal use of ‘$’ in
identifier - ‘id’ is not a keyword
err05.t.vdm:8: ‘$id’ undeclared
err05.t.vdm:9: Illegal greek letter code
after ‘i#’
err05.t.vdm:9: ‘j’ undeclared
err05.t.vdm:10: Illegal string constant
```

(End of file instead of closing ".)

```
err05.t.vdm:10: parse error
```

```
1  functions
2    f() r: bool
3    post r or n;
4
5    g(r: int)
6      r: bool
7    post true
```

```
> vdmpart err06.t.vdm h
err06.t.vdm:3: ‘n’ undeclared
err06.t.vdm:6: Redefinition of ‘r’
err06.t.vdm:5: ‘r’ previously defined here
err06.t.vdm: File does not declare implicit
function or operation ‘h’
```

```
1  functions
2    f1: int -> rat
3    f2() == 0
4
5  operations
6    o1: () ==> ()
7    o2() == skip
```

```
> vdmpart err07.t.vdm -1
err07.t.vdm:2: Names in function definition
must be identical: ‘f1’, ‘f2’
err07.t.vdm:6: Names in operation definition
must be identical: ‘o1’, ‘o2’
err07.t.vdm: File does not declare any
implicit functions or operations
```

```
1  operations
2    o() r: bool
3    ext wr x: int
4    pre (x + 1)
5        and r
6        and x
7    post r =
8        true
```

```
> vdmpart err08.t.vdm -1
err08.t.vdm:8: Result of expression is
boolean, should be integer
err08.t.vdm:7: Type of ‘r’ is not int
err08.t.vdm:6: Type of ‘x~’ is not bool
err08.t.vdm:4: Result of expression is
integer, should be boolean
```

(The operator “=” requires numeric operands. It would have to be replaced with “<=>” for the expression to be correct.)

```
1  functions
2    o(x, y: rat) r: rat
3    post r = x
4                /
5                  y
```

```
> vdmpart err09.t.vdm o
err09.t.vdm:4: Sorry, test case generation
not implemented
err09.t.vdm:4:  for this kind of expression
```

(Only integer division – the div operator – is supported.)

```
1  values
```

```
err10.t.vdm:2: parse error
```

(End of file instead of value definitions.)

5 Conclusion

In section 1, the nature of the problem that was to be solved in this Software Engineering Project was outlined, and in section 2, the VDM specification language was described in more detail in order to show the fundamental approach to test data generation taken by a program that was to be implemented as part of the project. The functionality of the program was also specified more accurately and broken down into several stages:

- Read the input characters of a VDM-SL specification and separate them into symbols in a scanner.
- In a parser, analyse these symbols and build a parse tree which represents the VDM-SL specification.
- In the parse tree, find the function/operation definition that is to be analysed and create a partition from the expressions it contains.
- Create equivalence classes from the partition and output them.

Prior to the design of these program components, the theoretical work the last two stages are based on – the Ph.D. thesis of Christophe Meudec – was analysed. Because of undesirable properties of the test data generation algorithm (in particular the fact that its memory requirements grow exponentially with the size of the input specification) and because of the possibility of a flaw in the algorithm itself, it was decided to modify the test data generation technique. The resulting algorithm is described in section 3.3.3. It turns out to be less efficient as far as the higher number of test cases that are created for an input is concerned. However, on the other hand its space complexity is far better and it allows that the test cases are generated in semi-random order.

The role of each of the components listed above was next defined in detail: Not only was its functionality given, but also its program interface (see section 3). Finally, all of the components were successfully implemented and integrated into one program. The components as well as the final program were tested as outlined in sections 4.5 and 4.6.

The tools and environment used for the implementation of the program proved to be well suited for the task. Previous experience with the scanner and parser generation tools flex and bison allowed the implementation to proceed relatively quickly. The C++ programming language used for the program was also a good choice as it could easily be interfaced to the C output of

the code generators, and is both both powerful and fast; because of the bad complexity of the problem, the latter property is not unimportant.

The program written for the project is suitable for further development – in fact, many parts have been written carefully in a way that makes later extension easy. First and foremost, the program is not ready for practical use in its current state – its output needs to be passed through an as yet unwritten solver to produce the final test data.

In addition to this, the support for VDM-SL is currently restricted to very few VDM operations and data types. Even though [Meudec98] states that it will not be possible to automate test data generation for all VDM constructs, support for a number of them could still be added to vdmpart. The most probable candidate for this kind of expansion are user-defined types, since they appear in almost every VDM-SL specification.

Furthermore, the partitioning and test data generation process as implemented in vdmpart can be enhanced in the way described on page 17 as soon as a solver is available.

Finally, it would also make sense to write a support program which can be used for the actual testing procedure. It will need to read in the test data produced by the extended vdmpart (because of the large amount of test data, preferably not from a file, but through a Unix pipe), instantiate the indicated state for a component of the program to be tested, execute that part of the program, and take notice of any errors and program crashes.

All in all, the program that was created during the course of the Software Engineering Project meets all requirements specified at the start of the project, and it is felt that it provides a sound basis for further work in the area of VDM-SL processing and test data generation.

A User Manual For vdmpart

A.1 Installation

The following is required to build the program from the source archive, `vdmpart-x.x.x.tar.gz` or `vdmpart-x.x.x.tar.bz2`:

- A Unix-like system, or CygWin under Windows.
- A C++ compiler conforming to the 1998 ISO C++ standard. While any such compiler should work, only GCC 2.95.x has been tested.
- A make utility, preferably GNU make.
- LaTeX 2ε to create the documentation and dvips to turn it into PostScript, as well as xdvi, ghostview or similar to view it.
- An implementation of awk, e.g. gawk, to run the test suite.
- Various other standard Unix tools: A shell, tar, gzip or bzip2, cmp, cat, touch etc.

Since the source archive comes with pre-generated source, the following programs are only needed when the source code is modified: bison, flex, autoconf, makedepend.

To compile the program:

- Untar the archive with one of the following commands, depending on which program it has been compressed with:

```
bzcat vdmpart-x.x.x.tar.bz2 | tar -xvf -
gzip -cd vdmpart-x.x.x.tar.gz | tar -xvf -
```

- Change into the directory that has been created and execute "`./configure`". This will perform some checks for features and programs available on your system, and in response alter the way the program is compiled.
- Execute the command "`make`" to compile vdmpart and to create its documentation. Since some of the code (in particular the parser) takes extremely long to compile with GCC 2.95, optimisation is switched off by default; if you want to compile the program with optimisation, use "`make X='-O2'`".

- Finally, if compilation has successfully completed, it is recommended that you execute "`make check`" to test whether vdmpart works correctly. This command will cause the program to be run on the files in the `examples` subdirectory.

A.2 Program Usage

The program vdmpart is a command line utility. It is invoked as

`vdmpart` ⟨*options*⟩ ⟨*input-file*⟩ ⟨*function-name*⟩

It reads a VDM-SL program from the specified *input-file* (the filename of which should have a "`.vdm`" extension) and writes test data information to standard output, or to a file specified with the `--testcases` switch.

Alternatively, the special value "`-`" can be given instead of the input filename, in which case the program reads from standard input.

Test data is generated for the implicit function or operation specified by the *function-name* argument.

The *input-file* and *function-name* parameters may only be omitted completely if one of the following two options is used:

`-h` or `--help`

Output information on the available option switches and exit immediately without processing the input.

`-v` or `--version`

Output the version number and exit immediately without processing the input.

The other options understood by the program are:

`-1` or `--first`

This option replaces the *function-name* parameter and causes the program to analyse the first implicit operation or function definition that the specified file defines.

`-t` or `--testcases`

Create test data and print it to standard output. Since this already is the default behaviour, the switch will only be used to redirect the output to a file using `--testcases=data.txt` or similar.

`-n` or `--no-testcases`

Suppress generation of test data.

`-p` or `--partition`

After the final partition for the specified function or operation has been generated, print it in a format suitable for processing by LaTeX. The output does not contain definitions for the macros used – a file which contains example definitions is included with the source distribution and in appendix A.3. Standard output is used, unless a filename is explicitly specified, e.g. with `--partition=part.tex`

`--debug-noterm`

When finished, do not exit, instead sleep forever. This allows for examination of the program's state, e.g. to find memory leaks.
This option is only available if the program has been compiled with `-DDEBUG`.

`--debug-noversion`

Do not include the version number of the program in the generated partition or test data. This switch is mainly useful in the following case: The test suite compares the program output with the saved output from a working vdmpart and considers a test to have failed if the output differs. The tests would fail whenever the version number changes, and would have to be adapted to the new version manually.

`--debug-parse`

Turn on output of debugging information in the generated parser.

`--debug-printtree`

"Pretty-print" the program using the parse tree once parsing has successfully finished. Standard output is used, unless a filename is explicitly specified, e.g. with `--debug-printtree=tree.txt`

A.3 LaTeX Macro Definitions For Use With vdmpart

The following file can be used to process the output generated with vdmpart's `--partition` switch with LaTeX. It is expected that the generated partition has been written to the file `partition.tex`.

```
\documentclass[10pt,fleqn]{article}
```

```
\newcommand{\vdmComb}[2]{\ensuremath{#1 \times #2}}
\newcommand{\vdmSet}[1]{\ensuremath{%
  \left\{\begin{array}{c}#1\end{array}\right\}}}
\newcommand{\vdmVar}[1]{\mbox{#1}}
\newcommand{\vdmOp}[1]{\mbox{\textsf{#1}}}

\addtolength{\oddsidemargin}{-1in}
\addtolength{\evensidemargin}{-1in}
\addtolength{\textwidth}{2in}
\addtolength{\topmargin}{-1in}
\addtolength{\textheight}{2in}

\begin{document}
\noindent\(\input{partition.tex}\)
\end{document}
```

B Example of Input/Output For a Small Problem

The command `vdmpart relations.vdm --first --partition=partition.tex` was used on the following input file:

```
functions

  fnc(x, y: int) r: bool
  post r <=> x > 0 or y < 5
```

The partition output by the program looks like this in ASCII:

```
% generated from 'fnc' in file 'relations.vdm' by vdmpart 0.9.0
\vdmSet{%
  \vdmComb{%
    \vdmSet{%
      \vdmVar{r}}%
  }{%
    \vdmSet{%
      \vdmComb{%
        \vdmSet{%
          (\vdmVar{x} = (0 + 1))\\%
          (\vdmVar{x} > (0 + 1))}%
      }{%
        \vdmSet{%
          ((\vdmVar{y} + 1) = 5)\\%
          ((\vdmVar{y} + 1) < 5)}%
      }%
      \\%
      \vdmComb{%
        \vdmSet{%
          (\vdmVar{x} = 0)\\%
          (\vdmVar{x} < 0)}%
      }{%
        \vdmSet{%
          ((\vdmVar{y} + 1) = 5)\\%
          ((\vdmVar{y} + 1) < 5)}%
```

```
        }%
        \\%
        \vdmComb{%
          \vdmSet{%
            (\vdmVar{x} = (0 + 1))\\%
            (\vdmVar{x} > (0 + 1))}%
        }{%
          \vdmSet{%
            (\vdmVar{y} = 5)\\%
            (\vdmVar{y} > 5)}%
        }%
      }%
    }%
    \\%
    \vdmComb{%
      \vdmSet{%
        (\vdmOp{not } \vdmVar{r})}%
    }{%
      \vdmComb{%
        \vdmSet{%
          (\vdmVar{x} = 0)\\%
          (\vdmVar{x} < 0)}%
      }{%
        \vdmSet{%
          (\vdmVar{y} = 5)\\%
          (\vdmVar{y} > 5)}%
      }%
    }%
  }%
```

When the partition is processed by LaTeX using the provided file `vdmpart-macros.tex`,

the following output is created:

$$\left\{ \begin{array}{l} \left\{ \text{ r } \right\} \times \left\{ \begin{array}{l} \left\{ \begin{array}{l} (\text{x} = (0+1)) \\ (\text{x} > (0+1)) \end{array} \right\} \times \left\{ \begin{array}{l} ((\text{y}+1) = 5) \\ ((\text{y}+1) < 5) \end{array} \right\} \\ \left\{ \begin{array}{l} (\text{x} = 0) \\ (\text{x} < 0) \end{array} \right\} \times \left\{ \begin{array}{l} ((\text{y}+1) = 5) \\ ((\text{y}+1) < 5) \end{array} \right\} \\ \left\{ \begin{array}{l} (\text{x} = (0+1)) \\ (\text{x} > (0+1)) \end{array} \right\} \times \left\{ \begin{array}{l} (\text{y} = 5) \\ (\text{y} > 5) \end{array} \right\} \end{array} \right\} \\ \left\{ \text{ (not r) } \right\} \times \left\{ \begin{array}{l} (\text{x} = 0) \\ (\text{x} < 0) \end{array} \right\} \times \left\{ \begin{array}{l} (\text{y} = 5) \\ (\text{y} > 5) \end{array} \right\} \end{array} \right\}$$

Finally, here is the set of test cases as output by vdmpart:

```
# generated from ‘fnc’ in file ‘relations.vdm’ by vdmpart 0.9.0
# 16 test cases
# 0
(r and ((x = (0 + 1)) and ((y + 1) = 5)))
# 8
(r and ((x = (0 + 1)) and (y = 5)))
# 4
(r and ((x = 0) and ((y + 1) = 5)))
# 12
((not r) and ((x = 0) and (y = 5)))
# 2
(r and ((x = (0 + 1)) and ((y + 1) < 5)))
# 10
(r and ((x = (0 + 1)) and (y > 5)))
# 6
(r and ((x = 0) and ((y + 1) < 5)))
# 14
((not r) and ((x = 0) and (y > 5)))
# 1
(r and ((x > (0 + 1)) and ((y + 1) = 5)))
# 9
(r and ((x > (0 + 1)) and (y = 5)))
# 5
(r and ((x < 0) and ((y + 1) = 5)))
# 13
((not r) and ((x < 0) and (y = 5)))
# 3
(r and ((x > (0 + 1)) and ((y + 1) < 5)))
# 11
(r and ((x > (0 + 1)) and (y > 5)))
# 7
```

```
(r and ((x < 0) and ((y + 1) < 5)))
# 15
((not r) and ((x < 0) and (y > 5)))
```

C LR(1) Grammar For VDM-SL

Below is a copy of the transformed grammar used by the parser generator, based on section 9 of [ISO93]. Keywords and other terminal symbols are printed in a **bold** typeface, terminal symbols consisting of non-alphabetic ASCII characters appear like "`this`", and nonterminals are *slanted*. Note that there are several symbols (for example "identifier") that are nonterminals in the ISO grammar, but terminals here, because they are handled by the scanner.

A number of ambiguities remain in this grammar. They are resolved using the precedence declarations for terminal symbols and rules that bison and yacc provide as an extension.

This LaTeX representation of the grammar has been automatically generated from the parser description file `src/parse.y` using the program yacc2tex, which was written especially for this purpose. It is supplied in the `doc` subdirectory of the source archive.

document = *openScope definition-block*
| *document definition-block* ;

definition-block = *type-definition-list*
| *state-definition*
| *state-definition* ***error***
| *value-definition-list*
| *function-definition-list*
| *operation-definition-list* ;

type-definition-list = **types** *type-definition*
| *type-definition-list* "`;`" *type-definition*
| *type-definition-list* "`;`" ***error***
| *type-definition-list* ***error*** ;

type-definition = *name* "`=`" *type*
| *name* "`=`" *type invariant*
| *name* "`::`" *field-list*
| *name* "`::`" *field-list invariant* ;

type-list = *type*
| *type-list* "`,`" *type*
| *type-list* ***error*** ;

type = "`(`" *type* "`)`"
| **bool**
| **nat**
| **nat1**
| **int**
| **rat**
| **real**

| **char**
| **token**
| ***lit-quote***
| **compose** *name* **of** *field-list* **end**
| *type* "|" *type*
| *type* "*" *type*
| "[" *type* "]"
| **set of** *type*
| **seq of** *type*
| **seq1 of** *type*
| **map** *type* **to** *type*
| **inmap** *type* **to** *type*
| *function-type*
| *type-variable-identifier*
| *name* ;

function-type = *type* "->" *type*
| "(" ")" "->" *type*
| *type* "+>" *type*
| "(" ")" "+>" *type* ;

discretionary-type = *type*
| "(" ")" ;

field = *name* ":" *type*
| *type* ;

field-list = ***empty***
| *field-list field* ;

state-definition = **state** *name* **of** *field-list* **end**
| **state** *name* **of** *field-list invariant* **end**
| **state** *name* **of** *field-list initialization* **end**
| **state** *name* **of** *field-list invariant initialization* **end**
| **state** *name* **of** *field-list initialization invariant* **end** ;

invariant = **inv** *pattern* "==" *expression* ;

initialization = **init** *pattern* "==" *expression* ;

value-definition-list = **values** *value-definition*
| *value-definition-list* ";" *value-definition*
| *value-definition-list* ";" ***error***
| *value-definition-list* ***error*** ;

value-definition = *pattern* "=" *expression*
| *pattern* ":" *type* "=" *expression* ;

function-definition-list = **functions** *openScope function-definition closeScope*
| *function-definition-list* ";" *openScope function-definition closeScope*

| *function-definition-list* “;” ***error***
| *function-definition-list* ***error*** ;

function-definition = *explicit-function-definition*
| *implicit-function-definition* ;

explicit-function-definition = *name* “:” *function-type name parameters-list* “==” *expression maybe-precondition*
| *name* “[” *type-variable-list* “]” “:” *function-type name parameters-list* “==” *expression maybe-precondition* ;

implicit-function-definition = *name parameter-type-list declName* “:” *type maybe-precondition* **post** *expression*
| *name* “[” *type-variable-list* “]” *parameter-type-list declName* “:” *type maybe-precondition* **post** *expression* ;

type-variable-list = *type-variable-identifier*
| *type-variable-list* “,” *type-variable-identifier* ;

type-variable-identifier = “@” *name* ;

parameter-type-list = “(” “)”
| “(” *pattern-type-pair-list* “)” ;

pattern-type-pair-list = *pattern-list* “:” *type*
| *pattern-type-pair-list* “,” *pattern-list* “:” *type* ;

parameters-list = *brack-maybe-pattern-list*
| *parameters-list brack-maybe-pattern-list* ;

maybe-precondition = ***empty***
| **pre** *beginPreCond expression endPreCond* ;

brack-maybe-pattern-list = “(” “)”
| “(” *pattern-list* “)” ;

operation-definition-list = **operations** *openScope operation-definition closeScope*
| *operation-definition-list* “;” *openScope operation-definition closeScope*
| *operation-definition-list* “;” ***error***
| *operation-definition-list* ***error*** ;

operation-definition = *explicit-operation-definition*
| *implicit-operation-definition* ;

explicit-operation-definition = *name* “:” *discretionary-type* “==>” *discretionary-type name brack-maybe-pattern-list* “==” *statement maybe-precondition* ;

implicit-operation-definition = *name parameter-type-list maybe-externals maybe-precondition* **post** *expression*
| *name parameter-type-list maybe-externals maybe-precondition* **post** *expression* **errs** *exception-list*

```
                                | name parameter-type-list declName ":" type maybe-externals
                                  maybe-precondition post expression
                                | name parameter-type-list declName ":" type maybe-externals
                                  maybe-precondition post expression errs exception-list ;

maybe-externals = empty
                | externals ;

externals = ext var-information
          | externals var-information
          | externals error ;

var-information = rd name-list
                | wr name-list
                | rd name-list ":" type
                | wr name-list ":" type ;

exception-list = name ":" expression "->" expression
               | exception-list name ":" expression "->" expression ;

expression-list = expression
                | expression-list "," expression
                | expression-list error ;

addto-expression-list = expression
                      | addto-expression-list "," expression
                      | addto-expression-list error ;

expression = "(" expression ")"
           | let openScope local-definition-list in expression closeScope
           | let openScope pattern in set expression in expression closeScope
           | let openScope name ":" type in expression closeScope
           | let openScope pattern2 ":" type in expression closeScope
           | let openScope pattern in set expression be st expression in expression closeScope
           | let openScope name ":" type be st expression in expression closeScope
           | let openScope pattern2 ":" type be st expression in expression closeScope
           | def openScope patternbind-expr-list in expression closeScope
           | if-expression
           | cases openScope expression ":" cases-alternatives end closeScope
           | cases openScope expression ":" cases-alternatives "," others "->" expression
             end closeScope
           | "+" expression
           | "-" expression
           | abs expression
           | floor expression
           | not expression
           | card expression
           | power expression
           | dunion expression
```

| **dinter** *expression*
| **hd** *expression*
| **tl** *expression*
| **len** *expression*
| **elems** *expression*
| **inds** *expression*
| **conc** *expression*
| **dom** *expression*
| **rng** *expression*
| **merge** *expression*
| **inverse** *expression*
| *expression* "+" *expression*
| *expression* "-" *expression*
| *expression* "*" *expression*
| *expression* "/" *expression*
| *expression* **div** *expression*
| *expression* **rem** *expression*
| *expression* **mod** *expression*
| *expression* "<" *expression*
| *expression* "<=" *expression*
| *expression* ">" *expression*
| *expression* ">=" *expression*
| *expression* "=" *expression*
| *expression* "<>" *expression*
| *expression* **or** *expression*
| *expression* **and** *expression*
| *expression* "=>" *expression*
| *expression* "<=>" *expression*
| *expression* **in set** *expression*
| *expression* **not in set** *expression*
| *expression* **subset** *expression*
| *expression* **psubset** *expression*
| *expression* **union** *expression*
| *expression* **merge** *expression*
| *expression* "\" *expression*
| *expression* **inter** *expression*
| *expression* "^" *expression*
| *expression* "++" *expression*
| *expression* **munion** *expression*
| *expression* "<:" *expression*
| *expression* "<-:" *expression*
| *expression* ":>" *expression*
| *expression* ":->" *expression*
| *expression* **comp** *expression*

| *expression* "**" *expression*
| **forall** *openScope bind-list* "&" *expression closeScope*
| **exists** *openScope bind-list* "&" *expression closeScope*
| **exists1** *openScope bind* "&" *expression closeScope*
| **iota** *openScope bind* "&" *expression closeScope*
| "{" "}"
| "{" *expression* "}"
| "{" *expression* "," *expression* "}"
| "{" *expression* "," *expression* "," *addto-expression-list* "}"
| "{" *expression* "|" *openComprehensionScope bind-list* "}" *closeScope*
| "{" *expression* "|" *openComprehensionScope bind-list* "&" *expression* "}" *closeScope*
| "{" *expression* "," "..." "," *expression* "}"
| "[" "]"
| "[" *expression-list* "]"
| "[" *expression* "|" *openComprehensionScope bind-list* "]" *closeScope*
| "[" *expression* "|" *openComprehensionScope bind-list* "&" *expression* "]" *closeScope*
| *expression* "(" *expression* "," "..." "," *expression* ")"
| "{" "|->" "}"
| "{" *map-enumeration-list* "}"
| "{" *expression* "|->" *expression* "|" *openComprehensionScope bind-list* "}" *closeScope*
| "{" *expression* "|->" *expression* "|" *openComprehensionScope bind-list* "&" *expression* "}" *closeScope*
| **mk_** "(" *expression-list* ")"
| **mk_identifier** "(" ")"
| **mk_identifier** "(" *expression-list* ")"
| **mu** "(" *expression* "," *record-modification-list* ")"
| *expression* "(" ")"
| *expression* "(" *expression* ")"
| *expression* "(" *expression* "," *addto-expression-list* ")"
| *expression* "." *name*
| *name* "[" *type-list* "]"
| **lambda** *type-bind-list* "&" *expression*
| **is_identifier** "(" *expression* ")"
| **identifier**
| *name* "~"
| *symbolic-literal* ;

symbolic-literal = ***lit-int***
| **true**
| **false**
| **nil**
| ***lit-char***

```
                | lit-string
                | lit-quote ;

patternbind-expr-list = pattern-bind "=" expression
                      | patternbind-expr-list ";" pattern-bind "=" expression
                      | patternbind-expr-list error ;

local-definition-list = local-definition
                      | local-definition-list "," local-definition
                      | local-definition-list error ;

if-expression = if expression then expression else expression
              | if expression then expression elseif-expression else expression ;

elseif-expression = elseif expression then expression
                  | elseif-expression elseif expression then expression ;

cases-alternatives = pattern-list "->" expression
                   | cases-alternatives "," pattern-list "->" expression
                   | cases-alternatives error ;

name-list = name
          | name-list "," name ;

map-enumeration-list = expression "|->" expression
                     | map-enumeration-list "," expression "|->" expression
                     | map-enumeration-list error ;

record-modification-list = name "|->" expression
                         | record-modification-list "," name "|->" expression
                         | record-modification-list error ;

name = identifier
     | mk_identifier
     | is_identifier ;

declName = name ;

state-designator = name
                 | state-designator "." name
                 | state-designator "(" expression ")" ;

statement = let local-definition-list in statement
          | let pattern in set expression in statement
          | let name ":" type in statement
          | let pattern2 ":" type in statement
          | let pattern in set expression be st expression in statement
          | let name ":" type be st expression in statement
          | let pattern2 ":" type be st expression in statement
          | def equals-definition-list in statement
          | "(" maybe-dcl-statement-list statement-list ")"
```

```
            | call-statement
            | skip ;

call-statement = name "(" ")"
               | name "(" expression-list ")"
               | name "(" ")" using state-designator
               | name "(" expression-list ")" using state-designator ;

equals-definition-list = pattern-bind "=" expression
                       | equals-definition-list ";" pattern-bind "=" expression ;

maybe-dcl-statement-list = empty
                         | maybe-dcl-statement-list name ":" type ";"
                         | maybe-dcl-statement-list name ":" type ":=" expression ";" ;

statement-list = statement
               | statement-list ";" statement ;

pattern = name
        | pattern2 ;

pattern2 = "_"
         | "(" expression ")"
         | symbolic-literal
         | "{" pattern-list "}"
         | pattern union pattern
         | "[" pattern-list "]"
         | pattern "^" pattern
         | mk_ "(" pattern-list ")"
         | name brack-maybe-pattern-list ;

pattern-list = pattern
             | pattern-list "," pattern
             | pattern-list error ;

pattern-bind = pattern
             | bind ;

bind = pattern in set expression
     | pattern ":" type ;

bind-list = multiple-set-bind
          | multiple-type-bind
          | bind-list "," multiple-set-bind
          | bind-list "," multiple-type-bind
          | bind-list error ;

multiple-set-bind = pattern-list in set expression ;

multiple-type-bind = pattern-list ":" type ;
```

type-bind-list = *pattern* ":" *type*
| *type-bind-list* "," *pattern* ":" *type*
| *type-bind-list* ***error*** ;

local-definition = *pattern* "=" *expression*
| *name* ":" *type* "=" *expression*
| *pattern2* ":" *type* "=" *expression*
| *name* ":" *function-type name parameters-list* "==" *expression maybe-precondition*
| *name* "[" *type-variable-list* "]" ":" *function-type name parameters-list* "==" *expression maybe-precondition*
| *name parameter-type-list declName* ":" *type maybe-precondition* **post** *expression*
| *name* "[" *type-variable-list* "]" *parameter-type-list declName* ":" *type maybe-precondition* **post** *expression* ;

openScope = ***empty*** ;

closeScope = ***empty*** ;

openComprehensionScope = ***empty*** ;

beginPreCond = ***empty*** ;

endPreCond = ***empty*** ;

D Program Code

The complete program source is submitted on floppy disc together with this document. It is also available online from http://www.in.tum.de/~atterer/uni/sep/

D.1 The SmartPtr template

The code excerpt below is included as an example of a complete compilation unit; `smartptr.h`. Only the initial copyright comment and some debugging code have been omitted. The SmartPtr template is described in detail in section 4.4.1.

```
struct SmartPtr_lockStatic;

struct SmartPtrBase {
  friend struct SmartPtr_lockStatic;
  SmartPtrBase() throw() : smartPtr_refCount(0) { }
  int smartPtr_refCount;
};
//---------------------------------------------------------------------

/* If static objects are accessed through smart pointers, ensure that
   there are no attempts to delete them, by defining a non-static
   SmartPtr_lockStatic(object), which MUST be DEFINED (not declared)
   AFTER the object being locked, in the SAME translation
   unit. Otherwise, order of construction is not defined. */
struct SmartPtr_lockStatic {
  SmartPtr_lockStatic(SmartPtrBase& obj) { ++obj.smartPtr_refCount; }
  ~SmartPtr_lockStatic() { }
};
//---------------------------------------------------------------------

// There are no implicit conversions from/to the actual pointer.
template<class X>
class SmartPtr {
public:
  typedef X element_type;
```

```
SmartPtr() throw() : ptr(0) { }
~SmartPtr() throw() { decRef(); }

// init from SmartPtr<X>
SmartPtr(const SmartPtr& x) throw() : ptr(x.get()) { incRef(); }
// init from SmartPtr to other type; only works if implicit conv. possible
template<class Y> SmartPtr(const SmartPtr<Y>& y) throw() : ptr(y.get()) {
  incRef();
}
// init from pointer
explicit SmartPtr(X* x) throw() : ptr(x) { incRef(); }

/* This one is necessary, the compiler will *not* generate one from
   the template below. */
SmartPtr& operator=(const SmartPtr& x) throw() {
  if (ptr != x.get()) { decRef(); ptr = x.get(); incRef(); }
  return *this;
}
template<class Y> SmartPtr& operator=(const SmartPtr<Y>& y) throw() {
  if (ptr != y.get()) { decRef(); ptr = y.get(); incRef(); }
  return *this;
}
template<class Y> SmartPtr& operator=(Y* y) throw() {
  if (ptr != y) { decRef(); ptr = y; incRef(); }
  return *this;
}

X& operator*()  const throw() { return *ptr; }
X* operator->() const throw() { return ptr; }
X* get()        const throw() { return ptr; }
X* release() throw() { // relinquish ownership, but never delete
  if (ptr != 0) --(ptr->SmartPtrBase::smartPtr_refCount);
  X* tmp = ptr; ptr = 0; return tmp;
}
void swap(SmartPtr& x) throw() { X* tmp = ptr; ptr = x.ptr; x.ptr = tmp; }
```

```
  bool isNull() const throw() { return ptr == 0; }

private:
  void incRef() throw() {
    if (ptr != 0) ++(ptr->SmartPtrBase::smartPtr_refCount);
  }
  void decRef() throw() {
    if (ptr != 0 && --(ptr->SmartPtrBase::smartPtr_refCount) <= 0)
      delete ptr;
  }
  X* ptr;
};
//--------------------

template<class X>
inline SmartPtr<X> makeSmartPtr(X* x) { return SmartPtr<X>(x); }

// only delete if count is zero
// 'deleteSmart(x);' is equivalent to '{ SmartPtr<X> tmp(x); }'
template<class X> // need template for 'delete ptr' to call the right dtor
inline bool deleteSmart(X* ptr) {
  if (ptr != 0 && ptr->SmartPtrBase::smartPtr_refCount <= 0) {
    delete ptr; return true;
  } else {
    return false;
  }
}

template<class X>
inline X* releaseSmart(X* ptr) {
  if (ptr != 0) --ptr->SmartPtrBase::smartPtr_refCount;
  return ptr;
}
//--------------------
```

```
template<class X> inline void swap(SmartPtr<X>& a, SmartPtr<X>& b) {
  a.swap(b);
}
template<class X>
inline bool operator<(const SmartPtr<X> a, const SmartPtr<X> b) {
  return a.get() < b.get();
}
template<class X>
inline bool operator>(const SmartPtr<X> a, const SmartPtr<X> b) {
  return a.get() > b.get();
}
template<class X>
inline bool operator<=(const SmartPtr<X> a, const SmartPtr<X> b) {
  return a.get() <= b.get();
}
template<class X>
inline bool operator>=(const SmartPtr<X> a, const SmartPtr<X> b) {
  return a.get() >= b.get();
};

// allow comparison with pointers
template<class X>
inline bool operator==(const SmartPtr<X> a, const X* b) {
  return a.get() == b;
}
template<class X>
inline bool operator==(const X* a, const SmartPtr<X> b) {
  return a == b.get();
}
template<class X>
inline bool operator!=(const SmartPtr<X> a, const X* b) {
  return a.get() != b;
}
template<class X>
inline bool operator!=(const X* a, const SmartPtr<X> b) {
```

```
    return a != b.get();
}
```

References

[ISO93] International Standards Organization. *Information Technology Programming Languages – VDM-SL*. First committee draft CD 13817-1, Document ISO/IEC JTC1/SC22/WG19 N-20, November 1993
Available online at
ftp://ftp.cs.uq.oz.au/pub/vdmsl_standard/
ftp://gatekeeper.dec.com/pub/standards/vdmsl/
ftp://ftp.imada.ou.dk/pub/vdmsl_standard/

[Meudec98] Christophe Meudec. *Automatic Generation of Software Test Cases From Formal Specifications*. Ph.D. thesis, The Queen's University of Belfast, May 1998
Available online as
http://www.geocities.com/CollegePark/Square/4148/research/thesis/thesis.zip
http://www.in.tum.de/~atterer/uni/sep/meudec-thesis.ps.gz

[Stroustrup97] Bjarne Stroustrup. *The C++ Programming Language*. 3rd edition, Addison Wesley, Reading (Massachusetts), 1997 (9th printing 1999)