

VDM — The Vienna Development Method

Andreas Müller

April 20, 2009

Bachelor thesis in "Formal Methods in Software Engineering", Johannes
Kepler University Linz, SS 2008.

Abstract

The Vienna Development Method is a formal language developed at the IBM laboratories in Vienna. First we give a short overview of the history of VDM from programming language description to VDM++. The language and its syntax are described in the following. Since the invention of VDM lots of tools have been developed. One of them is mural, a proof framework for VDM. We explain the basic features of mural and give a short example proof. The most important tool for VDM today is VDMTools which is still beeing developed. We give an overview of VDMTools features and present a detailed example of a VDM++ model in VDMTools. The example includes Java code generation from VDM.

Name: Andreas Müller
Matr.Nr.: 0555284
StKz.: 201
E-mail: a_m@gmx.at

Contents

1	Introduction	5
2	First Overview	6
2.1	Origins of the Vienna Development Method	6
2.2	Rigorous Specification and Proof	6
2.2.1	Rigorous Specification	7
2.2.2	Rigorous Proof	8
2.3	Formalization and Tools Support	9
3	The Language	9
3.1	Functions	10
3.1.1	Explicit Functions	10
3.1.2	Implicit Functions	11
3.2	Operations	13
3.2.1	Example: Simple Calculator	13
3.2.2	The State	15
3.3	Set Notations	15
3.3.1	The ”-set” Constructor	16
3.3.2	Partitions	16
3.4	Composite Objects	17
3.4.1	Defining Composite Objects	17
3.4.2	Creating Instances	18
3.4.3	Decomposing Objects	19

4	A Proof Framework for VDM — Mural	20
4.1	Constants and Expressions	21
4.2	Rules of Inference	22
4.3	Theories	22
4.4	Proofs	22
4.5	A Sample Proof	23
5	VDMTools	24
5.1	Main Features	24
5.2	VDM++	25
5.2.1	A Basic Class Outline	25
5.2.2	The Language	25
6	A VDMTools Example	27
6.1	Definition	27
6.1.1	Types	27
6.1.2	Instance Variables	28
6.1.3	Constructors	28
6.1.4	Operations	29
6.2	Screenshots	30
6.3	VDM to Java	30
6.4	Java to VDM	36
6.5	Other features	36
7	Conclusion	36

Bibliography	38
Index	39
A Complete Listings	39
A.1 Stack - VDM++ Model	39
A.2 Stack - Java Class	40

1 Introduction

The concept of formal methods describes a large number of scientific techniques for modelling and for rigorous checking of real-world systems. Formal methods are generally based on mathematical logic. There are many different approaches of formalization, but one of the longest established formal methods for development of computer-based systems is VDM — the Vienna Development Method.

The Vienna Development Method (VDM) is a collection of techniques for the modeling, specification and design of computer-based systems [5, page 454]. VDM has its roots in the IBM laboratories in Vienna in the mid-1970s. The corresponding standardized definition language is called VDM-SL. There is also an object-oriented extension of VDM, called VDM++. Today VDM is still very important and will be a topic at the 2009s 16th International Symposium on Formal Methods which will take place in Eindhoven (Netherlands) from November 2nd to November 6th, 2009. Of course there are also several industrial applications of VDM (especially VDMTools), for example Boeing used VDMTools for reverse engineering from Java back to VDM++ [9].

There are several books and articles on VDM. A huge collection of publications concerning VDM can be found on the VDM web-portal [10]. A few examples are the book from Derek Andrews and Darrel Ince, *Practical Formal Methods with VDM* [2] or D.J. Andrews (et al.), *The VDM Specification Language — Reading the Standard* [6]. This paper is mainly based on Peter Gorm Larsen's *VDM++ Tutorial* [9], on C. B. Jones' *Systematic Software Development Using VDM* [8] and on *Dines Bjørner's Logic Of Specification Languages* [5].

The paper is a summary of the history and the basic syntax of VDM. Its demand is to give a first introduction on VDM to the reader. The features of VDMTools are summarized and the final example of a VDM++ class explains the basics of VDM++ and VDMTools.

Chapter 2 gives a glimpse into the history of VDM. Chapter 3 describes the syntax of the VDM-SL and explains different types of functions and operations, and the use of sets. Chapter 4 introduces the Mural proof framework for VDM and includes a sample proof. Chapter 5 is about another tool for VDM: VDMTools. Chapter 6 describes an example VDM++ class. In Chapter 7 we present our conclusion.

2 First Overview

The following chapter gives a short overview about the historical development of VDM. It is mainly based on [5].

2.1 Origins of the Vienna Development Method

The origin of VDM can be found in the 1970s. VDM was at this time used in programming language description and compiler design. The main goal was to develop the language's fundamental features and to establish some formal semantics. One of the first notable uses of VDM was the attempt to give a formal definition of the PL/I language semantics. The notation used in this approach was called VDL — Vienna Definition Language.

Proofs were also an issue at the Vienna group. VDM was used to prove the equivalence of programming language concepts as part of compiler correctness arguments. At this stage, many different forms of arguments were explored. Although there were concerns about the quality and the style of the proofs, full formalization was not really needed until tool support became feasible.

1975 saw the dispersal of the Vienna group. This led to different approaches in the subsequent developments such that the modeling language, the methodology and the associated proof techniques evolved into several directions.

2.2 Rigorous Specification and Proof

In the 1980s, VDM experienced a shift from a definition language to a development "method"¹. The method became more and more standardized. This process was mainly guided by the so called VDM Symposia². At these meetings, first held in Brussels in 1987, different speakers reported their work with VDM. The presentations covered topics from denotational semantics to discharging of proof obligations, but tool support was not really an issue. The VDM standardization team stated in their report, that its work should be done by 1988 [1]. In reality, the standard was not approved by ISO till 1996.

¹The term "method" is — in context of VDM — used to describe a collection of development techniques.

²From the "VDM Symposia", the "FME" ("Formal Methods Europe) arose which developed into the "FM Symposia". The so called "International Symposium on Formal Methods" is held till today. [7]

2.2.1 Rigorous Specification

In Jones's book "Systematic Software Development Using CDM" [8] from 1986 many elements of VDM-SL can be found, which have not actually changed by now. The focus was on implicit style of operation specification. Below one can see a definition of a *biased queue*, taken from [8].

```

Queueb :: s : Qel*
         i : ℕ
where
inv-Queueb(mk-Queueb(s,i)) Δ i ≤ len s

ENQUEUE(e: Qel)
ext wr s: Qel*
post s =  $\bar{s} \hookrightarrow [e]$ 

DEQUEUE() e: Qel
ext rd s : Qel*
  wr i : ℕ
pre i < len s
post i =  $\bar{i} + 1 \wedge e = \bar{s}(i)$ 

```

A *biased queue* is a "biased" version of a queue because it stores a history of all elements ever stored in it. Adding an element to a biased queue appends the element to the end of the queue. Removing an element from a biased queue just returns the currently selected element and increases the selection-pointer. So no element is ever really deleted.

The model above now shows the definition of the described queue in the model-oriented VDM-SL. Such a model is constructed from basic types. In our case, these types are the *natural numbers* and *type constructors*. The type constructor X^* represents all finite sequences of elements of type X , in our example all sequences of *Qel*. So a value of type *Queueb* is constructed by a sequence of elements, representing the content of the queue and a natural number i which represents the last element taken from the queue.

Assignments to these variables are constrained by arbitrary predicates, so called *data type invariants*. Each pair of values s and i fulfilling the invariant represents a valid member of the type *Queueb*. For example the pair

$$s = [7, 8, 9] \text{ and } i = 2$$

represents a valid member of the type *Queueb*, whereas

$$s = [3, 6, 6, 6, 5] \text{ and } i = 42$$

does not represent any value of *Queueb*.

Operations are units of functionality capable of modifying the content of the state [5, page 457]. The given model describes two such operations: ENQUEUE and DEQUEUE. The operations are defined implicitly by post-conditions. Implicit definitions of course allow multiple implementations. The *ext*-statement is used as a *framing constraint* to define, which fields are accessed (rd) or changed (wr) by the operation. Note that the post-condition can only be fulfilled if all pre-conditions are satisfied. The model does not define what happens if an operation is applied to values that do not satisfy the pre-condition.

2.2.2 Rigorous Proof

VDM-SL with all its invariants, pre-conditions and post-conditions is a very complex and highly expressive language. Therefore it is not always possible to determine statically whether a model is consistent. This gives rise to *proof obligations*, i.e. conditions that have to be proved in order to ensure the adequacy of the model. An example for such an obligation is the *satisfiability obligation*: An operation has to yield a result satisfying the post-condition for every input that satisfies the pre-condition. The satisfiability obligation for the DEQUEUE-operation looks like this:

$$\begin{aligned} & \forall \bar{qb} \in \text{Queueb} : \text{pre} - \text{DEQUEUE}(\bar{qb}) \\ & \Rightarrow \exists qb \in \text{Queueb}, e \in \text{Qel} : \text{post} - \text{DEQUEUE}(\bar{qb}, qb, e) \end{aligned}$$

A proof for the condition above in style of [8] is shown below.

$$\begin{array}{ll} \text{from } \bar{qb} \in \text{Queueb}, \text{pre} - \text{DEQUEUE}(\bar{qb}) & \\ 1 \quad \text{let } i = \bar{i} + 1 & \\ 2 \quad \text{let } qb = mk - \text{Queueb}(\bar{s}, i) & \\ 3 \quad i < \text{len } \bar{s} & \text{h2} \\ 4 \quad i \leq \text{len } \bar{s} & \text{N,3,1} \\ 5 \quad \text{inv} - \text{Queueb}(qb) & 4,2,\text{inv} - \text{Queueb} \\ 6 \quad qb \in \text{Queueb} & 5,\text{Queueb} \\ 7 \quad \text{let } e = \bar{s}(i) & \\ 8 \quad e \in \text{Qel} & 7,4,\text{len} \\ 9 \quad i = \bar{i} + 1 \wedge e = \bar{s}(i) & \wedge - I(1,7) \\ 10 \quad \text{post} - \text{DEQUEUE}(\bar{qb}, qb, e) & \text{post} - \text{DEQUEUE}(9) \\ \text{infer } \exists qb \in \text{Queueb}, e \in \text{Qel} : \text{post} - \text{DEQUEUE}(\bar{qb}, qb, e) & \exists - I(6,8,10) \end{array}$$

The proof starts with the hypothesis (*from*) and ends with the conclusion (*infer*). Between these two lines each step either introduces a new definition, applies several rules or combines other lines. The justifications at the right of the lines show which rules were applied or which lines were combined. The numbers represent the line-numbers from the current proof and hn describes the n th hypothesis. Other rules, like the inference rule for conjunction shown below have to be defined somewhere else.

$$\begin{aligned} \wedge - I : & \frac{E_1; \dots; E_n}{E_1 \wedge \dots \wedge E_n} \\ \exists - I : & \frac{s \in X; E(s/x)}{\exists x \in X : E(x)} \end{aligned}$$

Proofs in VDM at this stage were primarily about finding weaknesses in models. They were often written on paper and meant to be read by other humans. The proof above is not formal. Some symbols are defined elsewhere, data types are not defined explicitly and justifications are just roughly specified. Thus it could not be checked by a machine.

2.3 Formalization and Tools Support

One of the first popular tools for VDM was *SpecBox*. It was developed by Bloomfield and Froome in the late 1980s. With these first attempts to give tool support, the semantic issues became more important. SpecBox already allowed some basic semantic checking, additionally to the syntax checking and pretty printing. [11]

Another important tool for VDM was the *VDM Toolbox* developed by the IFAD in Denmark. The *Toolbox* was based on META-IV, the executable part of the language. *VDM Toolbox* later evolved into the most popular tool for VDM: *VDMTools* (see chapter 5).

Because of growing industrial interest, additions to VDM were developed. The so called *Afrodite project* added some object-orientated and real-time extensions and invented VDM++.

3 The Language

As a mature and accepted language which has been used in a wide variety of applications, VDM supports lots of features for creation of formal models and proofs. The reader of this chapter is assumed to have the basic understanding of

formal methods, to know about propositions, predicates, operators and inference rules, and some basic knowledge on proofs. The content of this chapter is mainly based on [8].

3.1 Functions

Functions can be defined in two ways - either implicit or explicit. Both methods have advantages and disadvantages and are used in different kinds of situations.

3.1.1 Explicit Functions

An explicitly defined function consists of already known functions, operators, constants and parameters. The first line of an explicit function definition is the signature specifying the function's name, the input parameters and the output. The second line starts again with the name of the function, followed by a pair of brackets containing names for the input parameters so that they can be used later on. The Greek delta (Δ) is used as definition symbol. The equality sign ($=$) is not used to avoid confusion with predicates involving equalities (e.g. $square(2) = 4$). The following lines contain the direct definition.

The example below shows the definition for a function calculating the square of a given value.

$$\begin{aligned} &square: \mathbb{Z} \rightarrow \mathbb{N} \\ &square(i) \Delta i * i \end{aligned}$$

This definition only uses well known mathematical symbols. It is also possible to use conditions in explicit function. The next example is a definition for a function returning the absolute value.

$$\begin{aligned} &abs: \mathbb{Z} \rightarrow \mathbb{N} \\ &abs(i) \Delta \text{if } i < 0 \text{ then } -i \text{ else } i \end{aligned}$$

The use of the *let*-statement allows the user to introduce new local variables. The last example shows the definition of a function calculating the absolute value of the product of two given values.

$$\begin{aligned} &absprod: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{N} \\ &absprod(i,j) \Delta \text{let } k = i * j \text{ in (if } k < 0 \text{ then } -k \text{ else } k) \end{aligned}$$

3.1.2 Implicit Functions

An implicitly defined function does not specify how to calculate the solution, but what has to be calculated. The explicit definition can be seen as the implementation of the implicit specification. The most significant reason to give an implicit specification is simplicity. Implicit definitions are mostly shorter than explicit ones. For example it is easy to define a square root function implicitly, but it's much harder to implement an algorithm to approximately calculate it. However it's not always easier to give an implicit definition. The implicit specification of the algorithm for the UK income tax is not really shorter than the actual implementation.

Another reason to give an implicit definition is that they are easier to understand. What the square root function does is easy to understand. How it is done is much more complicated and in many cases also unimportant. A big advantage of implicit definitions is that they can not only yield a result for single values, but that it is also possible to evaluate the range of plausible results.

The danger of implicit with implicit specifications is, that they have to be very exact. The square root of a function can be either negative or positive. An implicit definition has to deal with this. It is only "correct" if it defines all properties the user wants to rely on.

The implicit definition of a function starts similar to an explicit one. The first line of the specification is the signature: names are given to argument and result and the names are followed by the type. Names are given as link to pre- and post-condition. The second contains the precondition and the third one the postcondition. Pre- and postcondition are arbitrary complex, boolean-valued functions, specifying the valid input values respectively the possible output. Since the possible input values for an implicit defined function are limited to values fulfilling the precondition, such functions can be seen as partial functions.

The following example illustrates the definition of an implicit function. It is the maximum function, that returns the biggest value of a given field.

$$\begin{array}{l} \text{maxs}(s: \mathbb{N}\text{-set}) \text{ } r: \mathbb{N} \\ \text{pre } s \neq \{\} \\ \text{post } r \in s \wedge \forall i \in s \cdot i \leq r \end{array}$$

The function *maxs* takes a finite set of natural numbers and returns a single natural number. In the signature the names *s* and *r* are assigned to these variables. The precondition is a boolean valued function with the following signature, that could — if needed — also be used outside the function.

$$pre-maxs: \mathbb{N}\text{-set} \rightarrow \mathbb{B}$$

The signature of the postcondition looks similar and can be seen below.

$$pre-maxs: \mathbb{N}\text{-set} \times \mathbb{N} \rightarrow \mathbb{B}$$

A test value for the function would be the field $\{6, 8, 1\}$. Before application of the function, the precondition has to be checked.

$$pre-maxs(\{6, 8, 1\}) \Leftrightarrow \mathbf{true}$$

The maximum value of the given field is of course 8. Obviously the postcondition holds for the field and the result.

$$post-maxs(\{6, 8, 1\}, 1) \Leftrightarrow \mathbf{true}$$

Implicit definitions can also include quantifiers. Quantifiers avoid recursions which would probably appear in direct definitions. The next example demonstrates the use of quantifiers in a definition. Also note that the precondition is always *true* and could be omitted. The function *gcd* gets two natural numbers greater than zero (\mathbb{N}_1) and returns their greatest common divisor. The used function *is-common-divisor* is a boolean function returning true, if *r* is common divisor of *i* and *j*.

$$\begin{aligned} &gcd(i: \mathbb{N}_1, j: \mathbb{N}_1) \text{ r: } \mathbb{N}_1 \\ &\quad \mathbf{pre} \ \mathbf{true} \\ &\mathbf{post} \ is\text{-common-divisor}(i, j, r) \wedge \neg \exists s \in \mathbb{N}_1 \cdot is\text{-common-divisor}(i, j, s) \wedge s > r \end{aligned}$$

As mentioned above, an implicit specified function may yield different values fulfilling the postcondition, depending on its implementation. The *arbs* function defined below is such a function. The postcondition only states, that the returned value has to be in the given field. Of course, depending on the implementation, each value of the field could be returned.

$$\begin{aligned} &arbs(s: \mathbb{N}\text{-set}) \text{ r: } \mathbb{N} \\ &\quad \mathbf{pre} \ s \neq \{\} \\ &\quad \mathbf{post} \ r \in s \end{aligned}$$

Summing up, implicit definition has lots of advantages.

- It is possible to describe several important directly.
- The set of results is described by the postcondition.
- The valid input-values are determined by the precondition.
- An algorithm solving the problem can be freely chosen.
- A name is assigned to the result via the function header.

If none of the above applies, a direct definition should be written. Of course, pre- and postconditions are themselves boolean functions and it is clear that explicit definitions have to be written sometimes or it would come to infinite regress.

3.2 Operations

While a function yields the same result for the same input values, an operation might give a different value each time because of a hidden state. The hidden state can for example be used as a recorder to save subsequent results. The state consists of all external variables an operation can access and change. For a function in any programming language this state is the set of the non-local variables; for a whole program it may be a database.

The *subsequent sum* shown below is an example of an operation. It sums up all given values and stores the current result in its state variable. The *double* function, which simply doubles the given value, is an example of a function.

sum(2) = 2	double(2) = 4
sum(2) = 4	double(2) = 4
sum(2) = 6	double(2) = 4
⋮	⋮

While *double* always yields 4 for the input value 2, the operation *sum* yields a different value (the subsequent sum) at every call.

3.2.1 Example: Simple Calculator

As an example of an operation, a simple calculator will be defined bit by bit below. The state in this case consists of a single value (*reg*) containing a natural number. This external variable is the link between the operations. The example is based on the example given in [8].

First we introduce a loading operation which simply loads the given value into the register:

$$\begin{array}{l} \text{LOAD}(i : \mathbb{N}) \\ \mathbf{ext\ wr} \text{ } reg : \mathbb{N} \\ \mathbf{post} \text{ } reg = i \end{array}$$

The first line is similar to a function definition: The name of the operation is followed by the parameters. Operations should be defined with capital letters.

The second line defines the operations access to the external variables. The *ext* keyword begins the line. The access modifiers *rd* (only read access) or *wr* (read and write access) followed by an external variable name and type define the rights the operation has on the given variable. The *load* operations of course needs write access to the register.

The postcondition is again a truth valued function of the parameters and the external variables. In this case, the simple postcondition only states that the value has been loaded into the register.

An operation which only needs read access to the register is *show* which yields the currently stored value of the register *reg*:

$$\begin{array}{l} \text{SHOW } () \text{ } r : \mathbb{N} \\ \mathbf{ext\ rd} \text{ } reg : \mathbb{N} \\ \mathbf{post} \text{ } r = \overline{reg} \end{array}$$

As the postcondition of the operation references to the value of *reg* before execution, *reg* is marked with an overline. The overline could also be omitted because the operation only has read access to *reg* and is not allowed to change it anyway.

$$\begin{array}{l} \text{SHOW } () \text{ } r : \mathbb{N} \\ \mathbf{ext\ rd} \text{ } reg : \mathbb{N} \\ \mathbf{post} \text{ } r = reg \end{array}$$

The equivalent definition of the show operations with write access is a bit more complicated:

$$\begin{array}{l} \text{SHOW } () \text{ } r : \mathbb{N} \\ \mathbf{ext\ wr} \text{ } reg : \mathbb{N} \\ \mathbf{post} \text{ } reg = \overline{reg} \wedge r = reg \end{array}$$

The two sample operations given above do not have any precondition i.e. the preconditions are assumed to be true. The divide operation however requires a precondition to prevent division by zero. This operation divides *reg* by a given value. It returns the result of the division and stores the remainder in *reg*.

$$\begin{array}{l} \textit{DIVIDE} \ (d : \mathbb{N}) \ r : \mathbb{N} \\ \quad \mathbf{ext \ wr} \ reg : \mathbb{N} \\ \quad \mathbf{pre} \ d \neq 0 \\ \mathbf{post} \ d * r + reg = \overline{reg} \wedge reg < d \end{array}$$

Note that identifiers in preconditions have no overlines although they refer to values before execution of the operation. The precondition is placed before the operation and the postcondition afterwards. So the undecorated values apply (in both cases) to the values the variables have at the current position.

3.2.2 The State

Just from knowing the state one cannot reconstruct the sequence of operation executions which led to this state because a single state can be reached in many different ways. For example, the state where *reg* has the value 1 can be reached in the following ways:

```
LOAD(1)
LOAD(7); DIVIDE(3)
```

The result of the next application of any operations only depends on the current value of the state and not on the history. In a mathematical way, each state creates the equivalence class on histories.

3.3 Set Notations

A *set* is in VDM a data structure which stores unordered, distinct elements³. The values contained by a set are bounded by braces. Sets can be formed in different ways:

- **Enumeration of their elements:** $\{9, 6, 42\}$
- **Set comprehension:** $\{i \in \mathbb{Z} | 1 \leq i \leq 3\} = \{1, 2, 3\}$

³Sets in VDM are basically equivalent to mathematical sets.

- **Intervals:** $\{i, \dots, k\} = \{j \in \mathbb{Z} | i \leq j \leq k\}$
- **Empty set:** $\{\}$

The number of items in a set (the cardinality) can be determined by the *card*-operator:

$$\begin{aligned} \text{card}\{\} &= 0 \\ \text{card}\{9, 6, 42\} &= 3 \end{aligned}$$

3.3.1 The "-set" Constructor

Another way of forming sets is the so-called "*-set*" constructor. This constructor can be applied to any known set and yields a set of all finite subsets of the given set. If the given set is infinite (e.g. the natural numbers) the number of these subsets can also be infinite. For finite sets it yields the power set.

Let \mathbb{B} be the set containing the boolean values *true* and *false*. The constructor would yield the following set:

$$\mathbb{B} - \text{set} = \{\{\}, \{\text{true}\}, \{\text{false}\}, \{\text{true}, \text{false}\}\}$$

3.3.2 Partitions

Partitions are sets fulfilling some special condition. So a set is called *partitioned*, if it can be into a set of disjoint subsets.

$$\text{Partition} = \{p \in (\mathbb{N} - \text{set}) - \text{set} \mid \text{inv} - \text{Partition}(p)\}$$

with

$$\begin{aligned} \text{inv} - \text{Partition} &: (\mathbb{N} - \text{set}) - \text{set} \rightarrow \mathbb{B} \\ \text{inv} - \text{Partition}(p) &\Delta \text{ is} - \text{prdisj}(p) \wedge \{\} \notin p \end{aligned}$$

and

$$\begin{aligned} \text{is} - \text{prdisj} &: (\mathbb{N} - \text{set}) - \text{set} \rightarrow \mathbb{B} \\ \text{is} - \text{prdisj}(ss) &\Delta \forall s_1, s_2 \in ss : s_1 = s_2 \vee \text{is} - \text{disj}(s_1, s_2) \end{aligned}$$

where *is-disj* means that the two given sets are disjunct i.e. they have no common elements.

An example for a partition can be seen below. The given set consists of two sets which are disjunct.

$$\begin{aligned} \{p_a, p_b\} &\subseteq \textit{Partition} \\ p_a &= \{\{1\}, \{2\}\} \\ p_b &= \{\{1, 2\}\} \end{aligned}$$

3.4 Composite Objects

Sets are not the only data structures used in VDM. Another kind of collections are multicomponent objects, so called *composite objects*. Composite objects can be compared with structures in C or records in Pascal. Such an object consists of various fields, each having a (different) value.

3.4.1 Defining Composite Objects

While classes of sets are defined by the *-set constructor*, composite objects are defined in another way:

```
compose [objectname] of
  [fieldname] : [fieldtype],
  ...
end
```

The objectname defines a name for the composite object. A composite object can consist of different fields whose name and type are set in the definition and separated by commas. An example for a composite object is a Date-Object which consists of two values, the day and the year.

```
compose Datec of
  day : {1, ..., 366},
  year : ℕ
end
```

It is not necessary to give a fieldname if the field is never accessed via its name. The definition for a Celcius-Object containing the temperature, may be changed

from

```
compose Celcius of  
     $v : \mathbb{R}$   
end
```

to

```
compose Celcius of  $\mathbb{R}$ 
```

Another shortcut for defining composite objects is the 'is composed of'-operator ($::$). A definition of a Date-Object mentioned above could be written as follows.

```
Datec ::  $day : \{1, \dots, 366\}$   
         $year : \mathbb{N}$ 
```

This abbreviation composes a Date-Object and gives it the name *Datec*.

3.4.2 Creating Instances

As distinguished from sets which are written using braces, instances of composite objects are created by *make*-functions. Given appropriate values for composite-objects fields, the make-function yields an instance with the respective values. The name of the make-function is composed by the prefix *mk*- and the name of the object one wants to create. The make-function for the Date-Object has the following structure: Given two values it creates an instance of Datec.

$$mk - Datec : \{1, \dots, 366\} \times \mathbb{N} \rightarrow Datec$$

An important property of a make-function is, that it creates *tagged* values. If one takes the Celcius-Object mentioned and a Fahrenheit-Object with the same fields (only one real number containing the temperature), the according make-functions will never yield the same values, even if the same temperature is set.

$$mk - Celsius(0) \neq mk - Fahrenheit(0)$$

Also if a make-function is called with two different values, it never yields the same instance.

$$mk - Celsius(0) \neq mk - Celsius(1)$$

3.4.3 Decomposing Objects

There are different ways of decomposing objects, one is by using so called *selectors*. Selectors are functions which are associated with the fields of a composite-object. They have to be applied to an object and to yield their current values. The signatures of the selectors for the day- and the year-field are

$$\begin{aligned} \text{day} &: \text{Datec} \rightarrow \{1, \dots, 366\} \\ \text{year} &: \text{Datec} \rightarrow \mathbb{N} \end{aligned}$$

So for example the following applies:

$$\begin{aligned} \text{day}(\text{mk} - \text{Datec}(7, 1979)) &= 7 \\ \text{year}(\text{mk} - \text{Datec}(117, 1989)) &= 1989 \end{aligned}$$

Another way of decomposing composite-object is the let-in-construct which binds a free variable. For example **let** $i = \dots$ **in** $\dots i \dots$ associates the value right of the equality sign to i and allows the use of i right of **in**. Obviously this can be used for decomposing objects. The example below shows the definition of a function which gets a *Datec* and yields a boolean value⁴.

$$\begin{aligned} \text{inv} - \text{Datec} &: \text{Datec} \rightarrow \mathbb{B} \\ \text{inv} - \text{Datec}(dt) &\Delta \text{ is} - \text{leapyr}(\text{year}(dt)) \vee \text{day}(dt) \leq 365 \end{aligned}$$

Alternatively the definition could be written as

$$\begin{aligned} \text{inv} - \text{Datec}(dt) &\Delta \\ \text{let } \text{mk} - \text{Datec}(d, y) = dt &\text{ in } \text{is} - \text{leapyr}(y) \vee d \leq 365 \end{aligned}$$

or even shorter as

$$\text{inv} - \text{Datec}(\text{mk} - \text{Datec}(d, y)) \Delta \text{ is} - \text{leapyr}(y) \vee d \leq 365$$

The cases-construct and the if-then-else-construct also allows decomposition of composite-objects and should be used, if there are several options to be resolved. The *norm-temp*-function yields the temperature in Celsius for a given Fahrenheit- or Celsius-value. Below it is defined with both, the if-then-else-construct and the cases-construct.

⁴In fact, this function may be an invariant for the Datec-Object, stating that a year either has 365 days or is a leap year.

```

norm - temp : (Fahrenheit  $\cup$  Celsius)  $\rightarrow$  Celsius
norm - temp(t)  $\Delta$ 
  if t  $\in$  Fahrenheit
    thenlet mk - Fahrenheit(v) = t in mk - Celsius((v - 32) *  $\frac{5}{9}$ )
  else t

```

```

norm - temp : (Fahrenheit  $\cup$  Celsius)  $\rightarrow$  Celsius
norm - temp(t)  $\Delta$ 
  cases t of
    mk - Fahrenheit(v)  $\rightarrow$  mk - Celsius((v - 32) *  $\frac{5}{9}$ )
    mk - Celsius(v)  $\rightarrow$  t
  end

```

Although there are several different ways of decomposing objects, it is rather obvious which way to use in which situation. The use of the cases-construct is clear and was mentioned above. Selectors should be used, if only a few fields of a multi-component object are accessed. On the other hand, if all fields of a composite-object are needed, the let-in-construct can be used.

4 A Proof Framework for VDM — Mural

Mural is an interactive specification support tool (SST) and a theorem-proving assistant (TPA) which has been developed as a part of the Alvey-funded IPSE 2.5 [12] project at Manchester University and the SERC Rutherford Appleton Laboratories [3]. *Mural* is meant for people with some knowledge about structuring a formal proof. The main features of this lightweight proofing assistant are

- **Bookkeeping:** The tool automatically stores all steps a user takes in a proof. Other axioms or theorems which have been proven before, can be saved and reused later.
- **Selection of applicable rules:** At any step of a proof, *Mural* suggests applicable rules. Of course a user can also select any other rule from the theory store.
- **Basic specification support environment:** *Mural* also includes a basic environment for creation of models and model-specific theories.
- **Theory store:** Axioms, definitions and theorems can be saved to a local theory store. This store initially includes the theorems and definitions for typed LPF⁵. Theorems which haven't been proved will be added as unproven conjectures to the theory store.

⁵Logic of Partial Functions

Despite all efforts on finding a source to get *Mural* I was not able to try it out myself. As popular the proving assistant may have been in the past, as untraceable it is nowadays. All examples in this chapter are taken from [5].

4.1 Constants and Expressions

Mural knows three different types of symbols:

- variables
- constants
- binders

A variable can hold an arbitrary value from a given range of values. Binders are used to introduce new bind-variables. Examples for binders are the quantifiers (\forall, \exists) .

Constants are constructors for values or types. An example for a constant would be the singleton sequence $([_])$ or the constructor for a finite set $(_ \text{-set})$. Each constant has a fixed arity. The arity describes the number of value and type arguments. The arity of the singleton sequence would be $(1,0)$ — one value argument, zero type argument — and the arity of the finite set constructor is $(0,1)$.

Expressions in *Mural* can be

- a variable symbol,
- a constant symbol,
- a binder or
- a notation of a subtype.

Constants always have to be used with the right number of arguments. Binders are used to introduce a new variable in another expression. The special notation for subtypes can be used to invent user defined types. As an example the subtype of all natural numbers smaller than ten is written as:

$$\langle \langle x : \mathbb{N} | x < 10 \rangle \rangle$$

4.2 Rules of Inference

Rules of inference in *Mural* are written in some sort of a Hilbert-style system. A rule starts with a name, followed by a line. Above the line stands the hypothesis and below the line the conclusion. Axioms look similar but have a trailing "Ax" after the horizontal line.

$$\begin{array}{c} - + 1 - form \frac{n : \mathbb{N}}{(n + 1) : \mathbb{N}} \\ \\ 0 - form \frac{}{0 : \mathbb{N}} Ax \end{array}$$

4.3 Theories

One feature of *Mural* is the so called *theory store*. A theory store collects all kind of theories in an inheritance structure. Such a theory is a collection of constant and binder definitions, axioms, derived results and also proofs. Theories can be used to limit the scope. In proofs, mural will only suggest inference rules from the current scope. Another important aspect of using theories is reusability i.e. already proven theorems can be reused in further proofs.

4.4 Proofs

A proof in *Mural* is structured into blocks, starting with *from* and ending with *infer*. A block is a sequence of arguments from hypothesis to conclusion with its own scope. Each of the inference steps consists of a line number, a formula and a justification. The justification always refers to an inference rule or to (un)folding of a syntactical definition.

If a proof is unfinished, it is marked as *unjustified*. If the user decides to proceed with proving an unjustified theorem, she/he is free to choose if he wants to work backwards from the goal or work forwards from the proposition. At each step, *Mural* recommends applicable rules of inference. Experienced users can also select rules which were not found by *Mural*. In this case the tool just does the pattern matching i.e. it fits the given values into the selected rule.

Each time a user want's to set up another hypothesis, a subproof is started, represented by another from-infer-block. Unproven blocks are also marked as *unjustified* until the user inserts a justification.

4.5 A Sample Proof

Assume, the user has entered the following inference rule:

$$\forall\text{-I} \frac{y : A \vdash_y P(y)}{\forall x : A \cdot P(x)}$$

The formula is added to the theory with status *unproved*. If the user selects it, the proof display opens:

```

from  $y : A \vdash_y P(y)$ 
      ...
infer  $\forall x : A \cdot P(x)$            <?? justify ??>

```

The conclusion line is flagged unjustified (<?? justify ??>). In this example the user decides to work backwards. He starts in the last line and applies the definition of \forall as $\neg\exists$. The justification tool updates the proof.

```

from  $y : A \vdash_y P(y)$ 
      ...
a       $\neg\exists x : A \cdot \neg P(x)$        <?? justify ??>
infer  $\forall x : A \cdot P(x)$            folding(a)

```

Notice, that the <?? justify ??>-marker has moved up a line and a justification was added to the last line. If the user applies a rule with a sequent hypothesis, the tool automatically starts a subproof. Mural handles the bookkeeping.

```

from  $y : A \vdash_y P(y)$ 
b      from  $z : A$ 
      ...
      infer  $\neg(\neg P(z))$            <?? justify ??>
a       $\neg\exists x : A \cdot \neg P(x)$         $\neg\exists\text{-I}(b)$ 
infer  $\forall x : A \cdot P(x)$            folding(a)

```

The <?? justify ??>-marker again moves up one line and a justification is added. The proof continues with forward reasoning inside the inner block. Note that the <?? justify ??>-marker disappears completely and the line numbers and justifications are updated.

```

from  $y : A \vdash_y P(y)$ 
1      from  $z : A$ 

```

1.1	$P(z)$	sequent h1 (1.h1)
	infer $\neg(\neg P(z))$	$\neg\neg\text{-I}(1.1)$
2	$\neg\exists x : A \cdot \neg P(x)$	$\neg\exists\text{-I}(1)$
	infer $\forall x : A \cdot P(x)$	folding(2)

The inference rules (axioms) used above can be found below.

$$\neg\exists\text{-I} \frac{x : A \vdash_x \neg P(x)}{\neg\exists y : A \cdot P(y)} Ax$$

$$\neg\neg\text{-I} \frac{e}{\neg\neg e} Ax$$

5 VDMTools

VDMTools is a Toolkit for development of model-oriented specifications in VDM-SL and VDM++, an object oriented extension of VDM-SL. VDMTools supports lots of different features, from basic syntax-checking to generation of models from Java-code. In 2004 VDMTools was sold to CSK Corporation, Japan. The tool is still being developed: the latest version 8.1 was released in Mai 2008! The use of VDMTools with VDM++ is described in the following subsections. The main sources of this chapter are [4] and [9].

5.1 Main Features

The toolkit has lots of usefull features from syntax checking to code generation:

- **Syntax checking:** The syntax-checker verifies whether the syntax of the selected files matches the VDM++ language specifications. If the check passes, it gives access to the other features of VDMTools.
- **Type checking:** The type-checker tests mis-uses of values and operators and can also show places, where runtime errors may occur.
- **Code generation:** VDMTools is able to generate a fully executable code for about 95% of all VDM++ constructs. Code generation is available for Java and C++.
- **Specification manager:** A manager-window displays all classes and files in the specification. It also shows the status for each file.
- **Interpreter and Debugger:** VDMTools allows to execute all executable VDM++ constructs. Debugging is also supported.

- **Integrity examiner:** It extends the static checking capabilities of VDM++. The tool scans through all sources to find possible inconsistencies or integrity violations. The examiner creates expressions which should evaluate to true. If they evaluate to false, there may be a problem.
- **Rose-VDM++ Link:** The Rational-Rose-Link provides a bi-directional link between Rational Rose (UML) and the Toolbox (VDM++).
- **Java to VDM++ Translator:** It is possible to generate a VDM++ specification from a Java application. The generated model can be examined at VDM++ level.
- **Several input types:** Models for VDMTools can be written either with Microsoft Word (RTF) or in Latex. Plain text is also possible but is not recommended.

5.2 VDM++

This section examines the basic features of VDM++. It is mainly based on [9].

5.2.1 A Basic Class Outline

VDM++ is an object-oriented specification language and an extension of VDM-SL. VDM uses classes to describe specification models. A VDM++ class consists of several parts (Figure 1). Instance variables, operations and functions can be found at the beginning of a class. VDM++ also allows the use of threads which communicate via shared objects. The concurrency behavior can be specified in the thread and the sync section of a class.

5.2.2 The Language

- **Constructors:** As known from programming languages, a VDM++ class can have several constructors.
- **Access Modifiers:** Class members may be *private*, *public* or *protected*. Default access is *private*. *static* members are also possible.
- **Instance Variables:** Variables can be used to represent model attributes or model associations.
- **Type definitions:** VDM++ supports several basis types (e.g. Boolean, Numeric, Characters,...) and does also allow compound types (e.g. Set types, Map types, Union types,...). Type invariants may also be added.

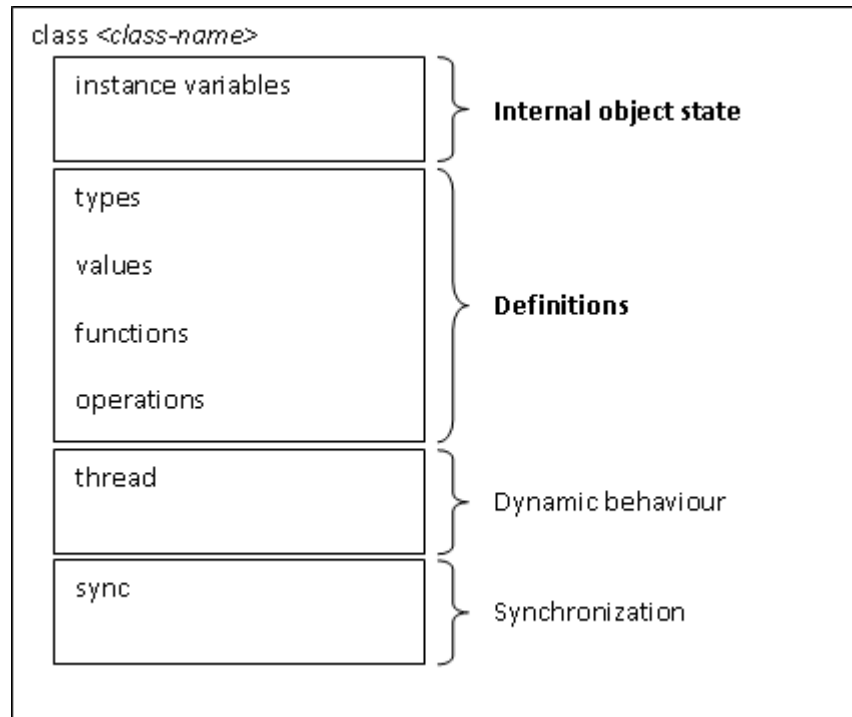


Figure 1: VDM++ class outline

There are several predefined operations for the different types such as sequence concatenation or set membership.

- **Functions:** Functions can be defined explicitly or implicitly. Implicitly defined functions cannot be executed by the interpreter.
- **Expressions:** Different common expressions such as if-then-else are available in VDM++.
- **Operations:** Operations may also be implemented explicitly or implicitly.
- **Threads:** Threads can be defined using the *thread* section of the class.

6 A VDMTools Example

As an example for a VDM++ Model in VDMTools a *Stack*-Class⁶ will be implemented. The *Stack* consists of an instance variable and different operations and constructors. To show the usage of types, a type is also included. The shown example is based on the Stack-Class which was taken from the examples delivered with VDMTools.

6.1 Definition

At first the Stack-Class which consists different areas has to be defined .

```
class Stack

    types
        ...

    instance variables
        ...

    operations
        ...

end Stack
```

6.1.1 Types

The example stack is able to store several *StackItems*. These *StackItems* are represented by integer values and defined in the *types*-section of the class.

```
types
    public StackItem = int;
```

An advantage of defining a particular type for *StackItem* is, that it is easier to switch the data type supported by the stack. For example if the stack should store real numbers instead of integers, only *StackItem* has to be changed to *real*.

⁶A stack is a data structure which accesses its elements Last-In-First-Out. So the least added item can be removed or read.

6.1.2 Instance Variables

The internal state of the Model-Class is described by instance variables defined in the according section of the class. For the stack example a sequence to store the items is needed.

```
instance variables
  stack : seq of StackItem;
```

6.1.3 Constructors

To initialize the instance variables with appropriate values constructors are implemented. Constructors are defined in the *operations*-section of the class. The default constructor (which takes no arguments) initializes an empty stack.

```
operations
  public Stack: () ==> Stack
    Stack() ==
      (stack:=[]);
```

As seen above, constructors are operations which have same name as the class and return a new instance of the class. Several constructors can be defined using operation overloading. For example the following constructor takes a *StackItem* (in our example an integer value) and initializes a *Stack* containing the given element.

```
public Stack: (StackItem) ==> Stack
  Stack(val) ==
    (stack:=[val]);
```

The last constructor takes a predefined sequence of *StackItems* and simply initializes the stack with it.

```
public Stack: (seq of StackItem) ==> Stack
  Stack(init) ==
    (stack:=init)
```

6.1.4 Operations

The basic stack-operations are *push* and *pop*. *Push* puts a given item on the top of the stack, *pop* removes the top-item.

```
public Push : StackItem ==> seq of StackItem
Push(val) ==
  (stack := [val]^stack;
   return stack)
post stack = [val]^stack~;
```

The *Push*-operation appends the given *StackItem* to the sequence and returns the new stack. Also a post-condition for the *Push*-operation is defined. This post-condition is added to the *Integrity-Checker* and must be checked by the user.

```
public Pop : () ==> StackItem
Pop() ==
  def res = hd stack in
    (stack := tl stack;
     return res)
pre stack <> []
post stack~ = [RESULT]^stack~;
```

The *Pop*-operation removes the first element from the stack and returns it. At first the head element is stored in *res*. The *hd*-command is a keyword returning the head-element of the sequence. Then the head is removed by calling *tl*, which returns the tail of the list. Finally *res* is returned.

Additionally to the post-condition, the *Pop*-operation is equipped with a pre-condition. The pre-condition states that the stack may not be empty before calling *Pop*. Pre-conditions are important when using the VDM++-To-Java function of VDMTools, as a boolean valued function is created for each pre-condition, which allows the programmer to check if all pre-conditions are fulfilled before calling the function. If a function is called, the pre-conditions are also checked automatically and if they are not satisfied an exception is thrown.

Finally the additional functions *Top* (returns the head of the Stack, but does not remove it) and *Reset* (empties the stack) can be found below.

```
public Top : () ==> StackItem
Top() ==
```

```

        return (hd stack)
pre stack <> []
post stack = stack~;

public Reset : () ==> ()
Reset () ==
    stack := [];

```

6.2 Screenshots

This section contains several screenshots of VDMTools while working with the example class. While Figure 2 shows the project view, Figure 3 shows the class view of the project. On the right side the source window can be found. It only displays the code and does not allow editing. The log window on the bottom left side of the window logs all past events. In the current example the syntax- and type checks were successful and all possible outputs were created (Java, C++, Pretty Print). This can be seen in the class view.

Figure 4 shows the Integrity examiner. All possible integrity problems are listed and can be marked as checked.

Figure 5 shows the Interpreter windows. The *create* command creates a new instance of class *Stack*. The *print command* is used to test the operations.

Figure 6 shows the Error List. The position of the selected error is automatically highlighted in the source window. Furthermore a detailed error description is displayed below.

6.3 VDM to Java

After syntax- and type-checking, VDMTools is able to create a Java file from the given model. After adding the included jar-Files to the Java-classpath the created file compiled without any errors. All functions worked properly and no further problems occurred. If any post-condition for a called function is not fulfilled, a *VDMRunTimeException* is thrown. The sequence of StackItems is defined as an object of type *java.util.Vector*. The type *StackItem* is automatically translated to *java.lang.Integer*. The whole content of the created file can be found in Appendix A.2.

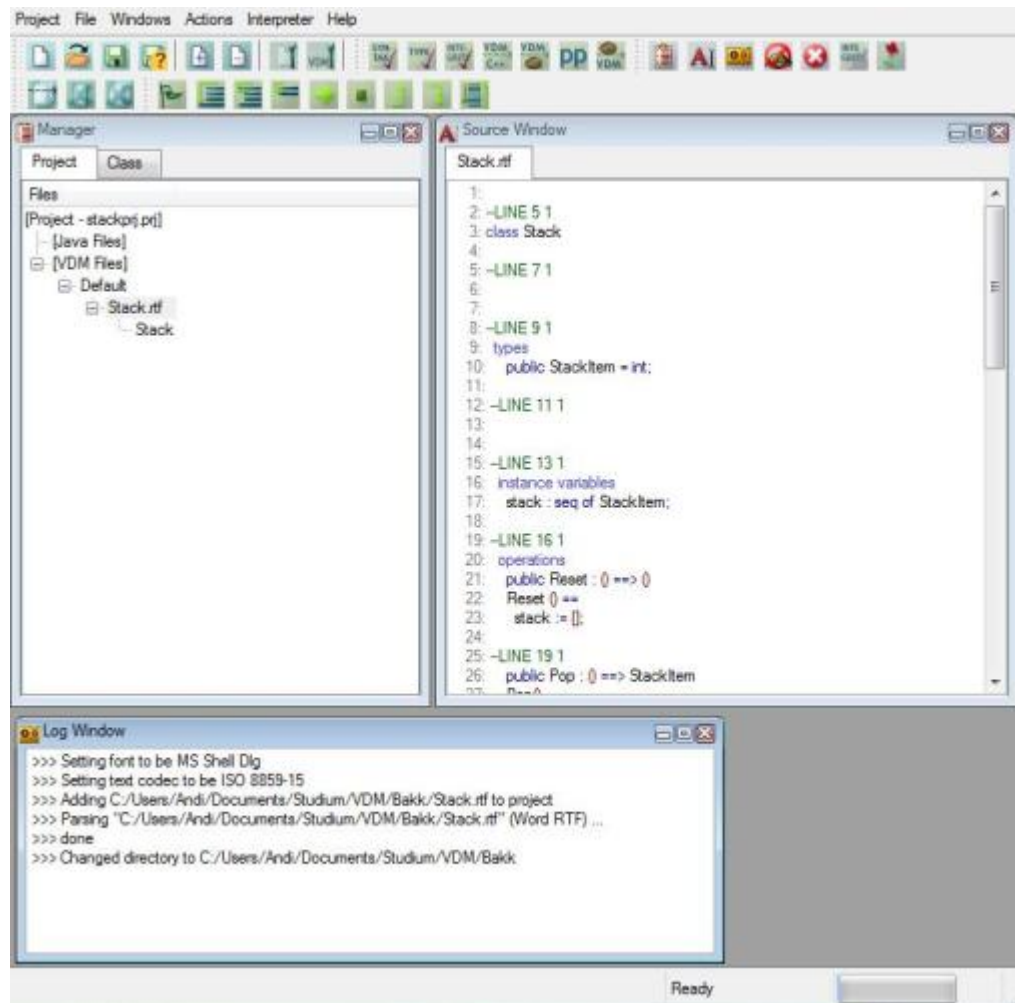


Figure 2: Project view

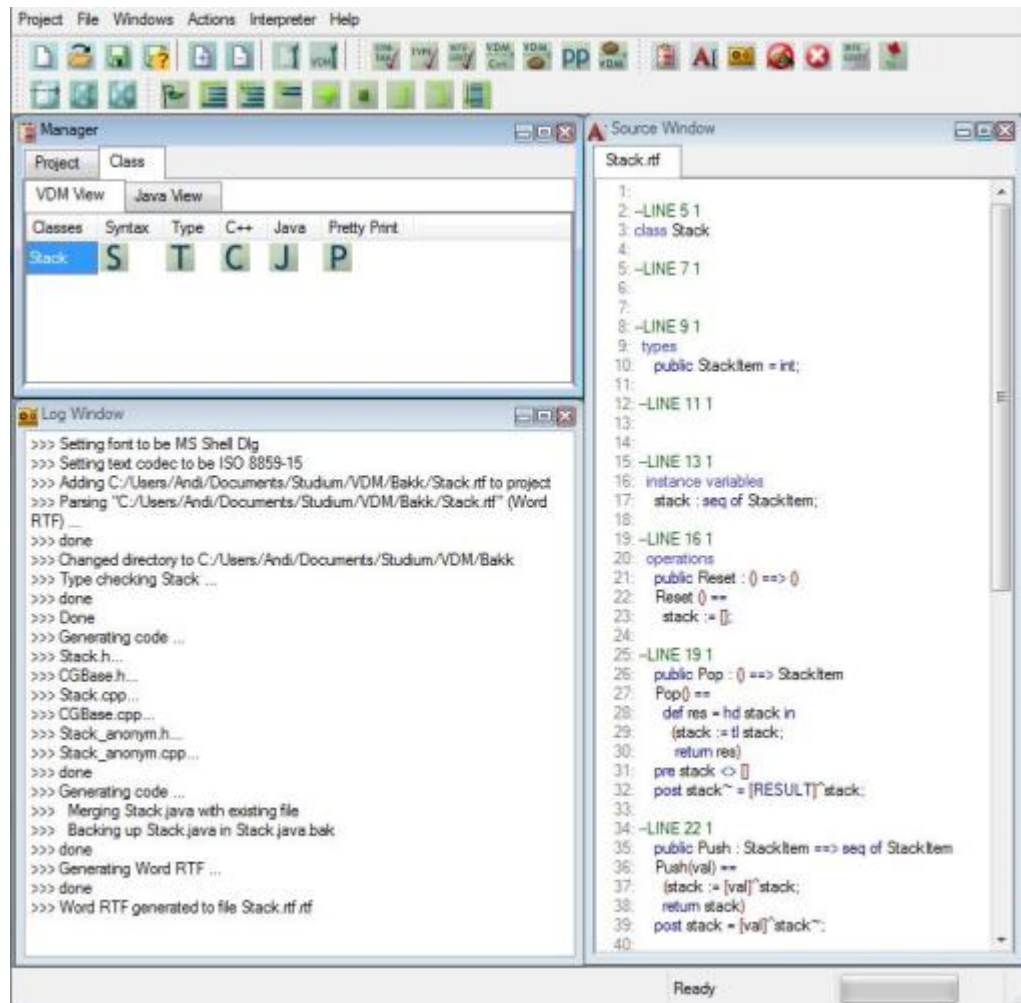


Figure 3: Class view

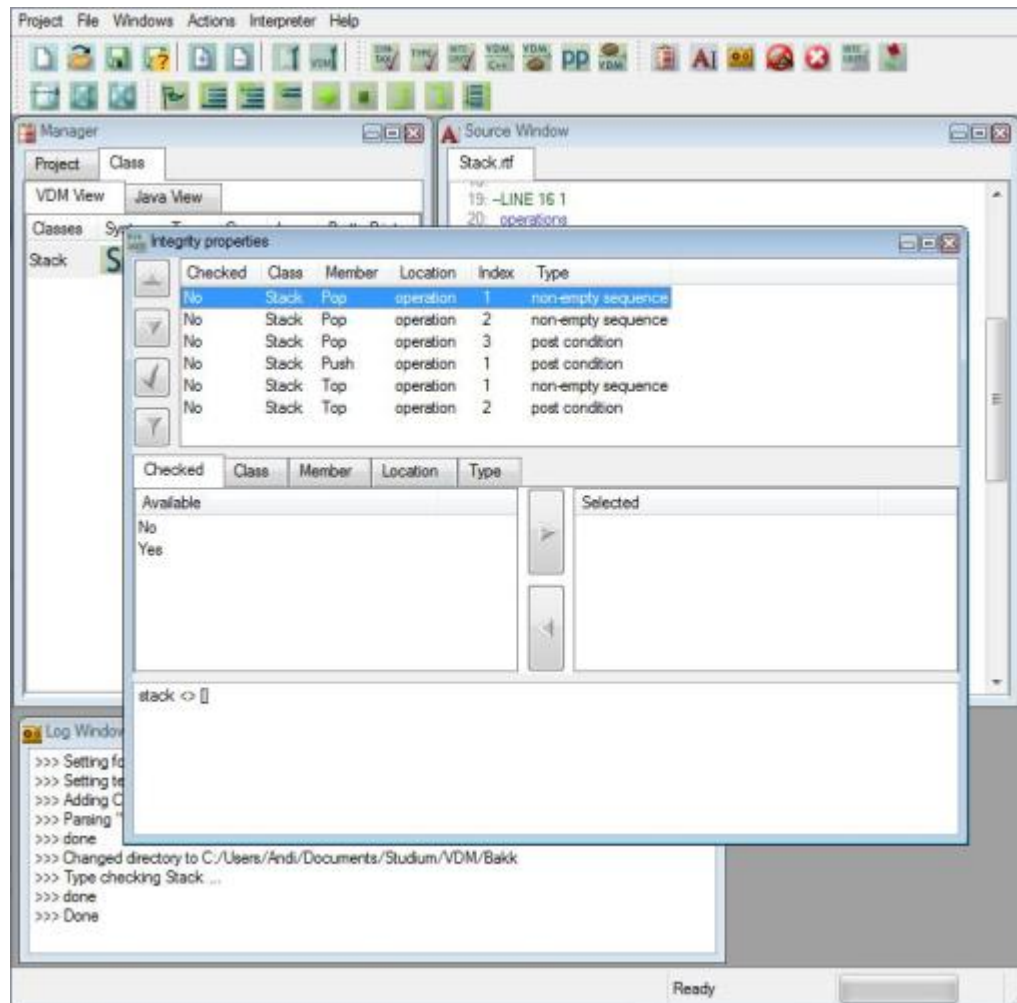


Figure 4: Integrity Properties

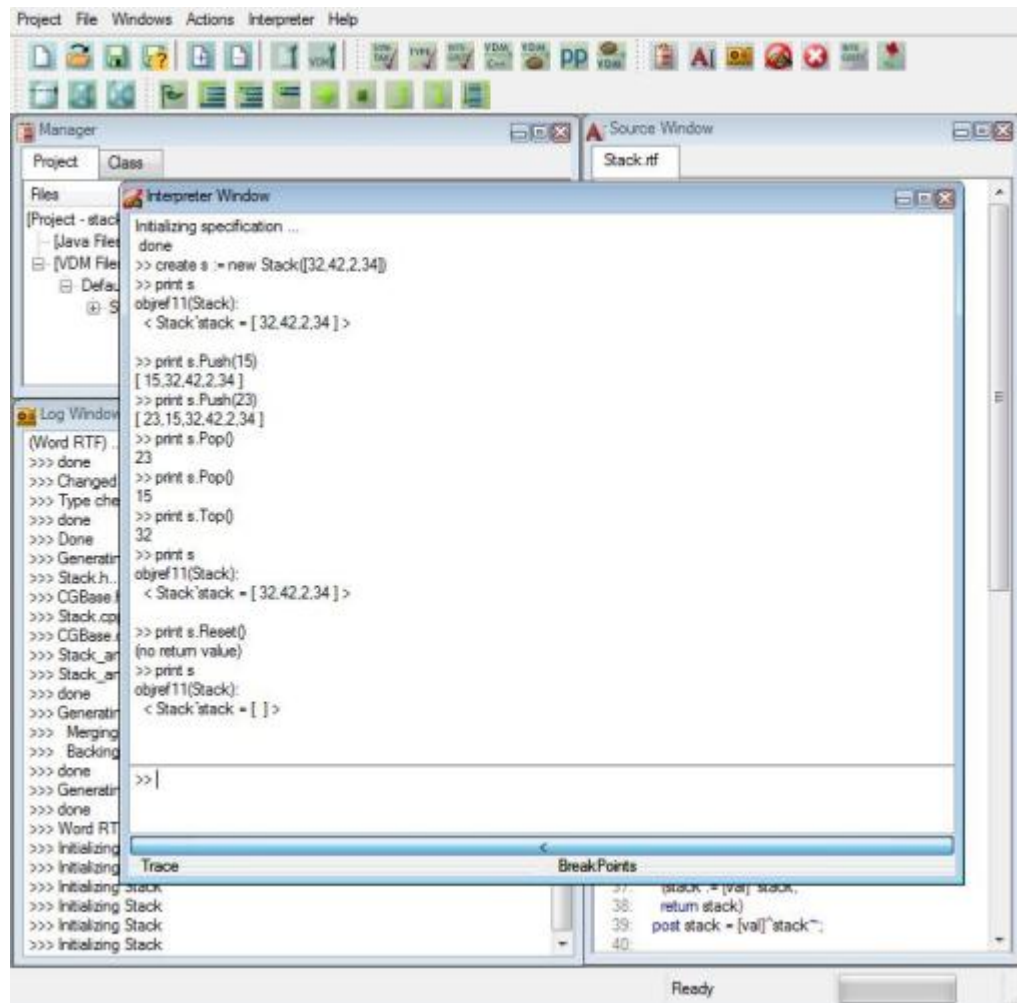


Figure 5: Interpreter

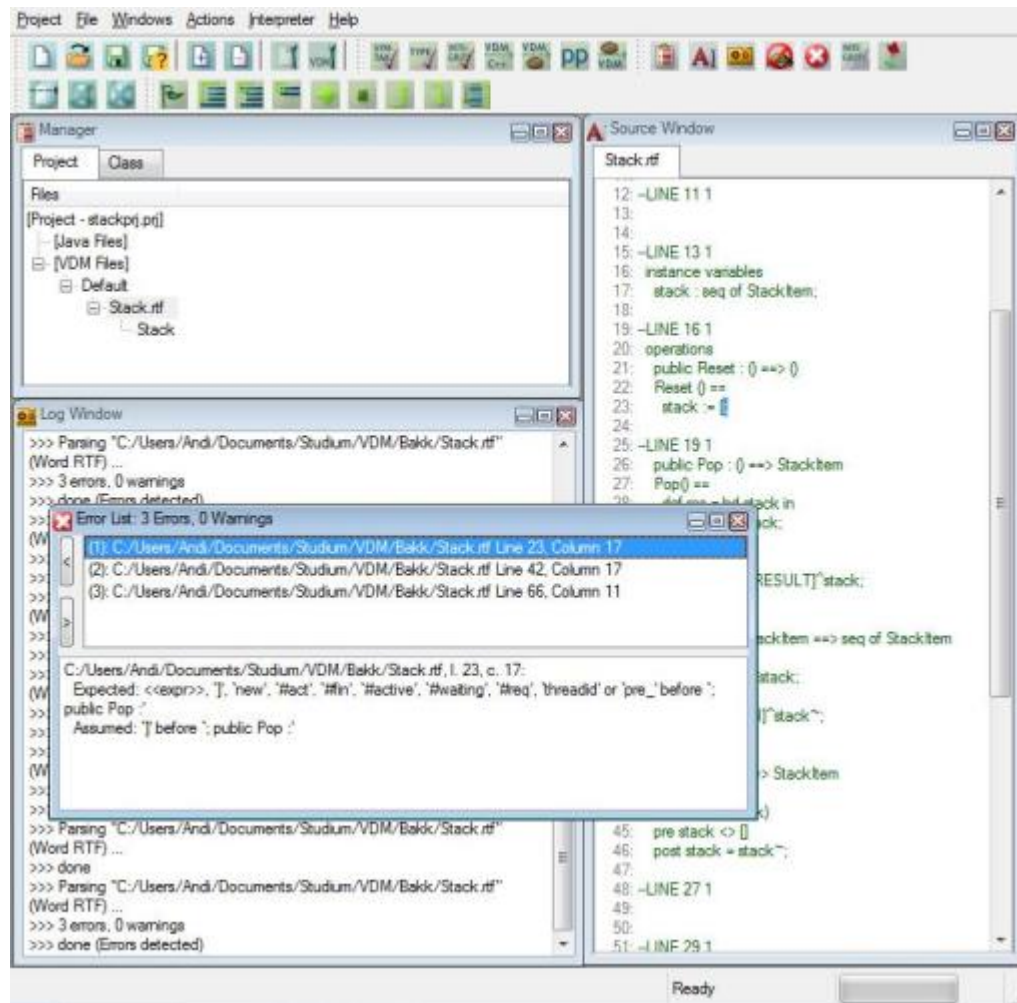


Figure 6: Errors

6.4 Java to VDM

While trying to convert the created Java file back to VDM++, VDMTools crashed with different runtime-errors (Figure 7). Although the transformation worked for the provided examples, it was not possible to convert the *Stack.java* back to VDM.

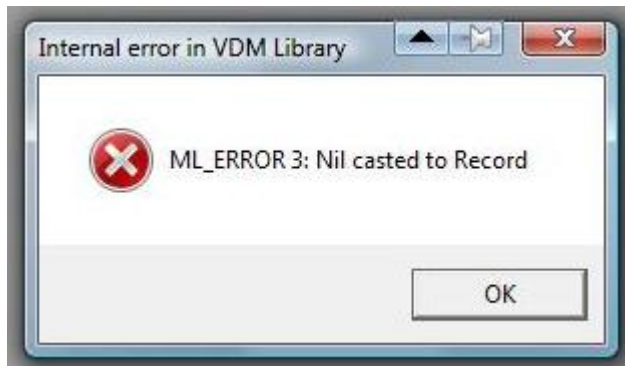


Figure 7: Java to VDM - Error

6.5 Other features

The PrettyPrint feature creates a RTF-file for use with Microsoft Word in which all keywords are highlighted. The VDM-To-C++ feature creates a compilable C++ file.

The stack example also worked fine with the VDMTools interpreter. It is possible to create various Stack objects, calling the different constructors. All operations can be called, but the interpreter does not check any pre-conditions and crashes if an error occurs.

7 Conclusion

In the history of formal methods, the Vienna Development Method is one of the longest-established formal methods. During its long lifetime, many different tools, standards and formalizations arose and disappeared. The most popular tool for VDM today (VDMTools) is a rather useful tool for development of formal models in VDM++ or VDM-SL.

The code creation features of VDMTools for Java and C++ are very helpful and work properly. Syntax- and type-checking ensure syntactically correct models and the Integrity-Examiner provides integrity-conditions which have to be proven or at least observed. The interpreter and especially the debugger are a simple way of testing the model and make it easy to discover the error source.

A disadvantage of VDMTools is the lack of usability. There is no internal editor for the models, so the user always has to use for example Microsoft Word to change the specification. The Error List cannot be emptied and so it is hard to see which errors are new and which have already been fixed.

VDM-SL and VDM++ are rather intuitive and well documented formal specification languages. VDM++ is based on VDM-SL and used for specification of object oriented models. Both languages provide a large pool of features, from simple data type definitions to multithreading and thread synchronisation.

The Vienna Development Method has a great community. The VDM portal webpage [10] lists 1000 different publications about VDM and the VDM forum is a simple way to communicate with other people working with VDM.

I was not able to test the mural proof framework for VDM or any other tool that supports guided proofs and cannot say much about their functionality. Since programs like VDMTools are able to generate conditions which have to be proven to ensure correctness of the model, these conditions have to be proved elsewhere or by hand.

References

- [1] D. Andrews. Report From The BSI Panel For The Standardisation Of VDM (ist/5/50). In *VDM '88 VDM — The Way Ahead*. Springer Berlin/Heidelberg, 1988.
- [2] Derek Andrews and Darrel Ince. *Practical Formal Methods With VDM*. McGraw Hill, September 1991. ISBN 0-07-707214-6.
- [3] Morten Elvang-Goransson Bob Fields. A VDM Case Study In Mural. *IEEE Transachons on Software Engineering*, 18(4):279–295, 1992.
- [4] CSK SYSTEMS CORPORATION. *VDMTools User Manual (VDM++) ver.1.2*, 2008. <http://www.vdmtools.jp/en/>.
- [5] Martin C. Henson Dines Bjørner. *Logic Of Specification Languages*. Springer, 2008.
- [6] B.S. Hansen P.G. Larsen N. Plat et.al D.J. Andrews (ed), H.Ûruun. *The VDM Specification Language — Reading The Standard*. Provionally accepted by Pretice-Hall for publication, 1995.
- [7] Fm 2008: 15th International Symposium On Formal Methods, 2008. <http://www.fm2008.abo.fi/>.
- [8] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 2nd edition, 1986.
- [9] Peter Gorm Larsen. VDM++ Tutorial At FM 06. Handout, 2006. <http://fm06.mcmaster.ca/VDM++>
- [10] VDM Portal, 2009. <http://www.vdmportal.org/>.
- [11] Adelard Richard Moore, Peter Froome. Mural And Specbox. In *VDM'91 Formal Software Development Methods*. Springer Berlin / Heidelberg, 1991.
- [12] R. A. Snowdon. An Introduction To The IPSE 2.5 Project. In *Software Engineering Environments*, pages 13–24. Springer Berlin / Heidelberg, 1990.

A Complete Listings

The following listings describe the example presented in Chapter 6.

A.1 Stack - VDM++ Model

```
class Stack

  types
    public StackItem = int;

  instance variables
    stack : seq of StackItem;

  operations
    public Reset : () ==> ()
    Reset () ==
      stack := [];

    public Pop : () ==> StackItem
    Pop() ==
      def res = hd stack in
        (stack := tl stack;
         return res)
    pre stack <> []
    post stack~ = [RESULT]^stack;

    public Push : StackItem ==> seq of StackItem
    Push(val) ==
      (stack := [val]^stack;
       return stack)
    post stack = [val]^stack~;

    public Top : () ==> StackItem
    Top() ==
      return (hd stack)
    pre stack <> []
    post stack = stack~;

    public Stack: () ==> Stack
    Stack() ==
      (stack:=[]);
```

```

    public Stack: (StackItem) ==> Stack
        Stack(val) ==
            (stack:=[val]);

    public Stack: (seq of StackItem) ==> Stack
        Stack(init) ==
            (stack:=init)

end Stack

```

A.2 Stack - Java Class

```

//
// THIS FILE IS AUTOMATICALLY GENERATED!!
//
// Generated at Thu 05-Mar-2009 by the VDM++ to JAVA Code Generator
// (v8.1 - Wed 19-Mar-2008 09:16:54)
//
// Supported compilers:
// jdk1.4
//

// ***** VDMTOOLS START Name=HeaderComment KEEP=NO
// ***** VDMTOOLS END Name=HeaderComment

// ***** VDMTOOLS START Name=package KEEP=NO
// ***** VDMTOOLS END Name=package

// ***** VDMTOOLS START Name=imports KEEP=NO

import jp.co.csk.vdm.toolbox.VDM.*;
import java.util.*;
import jp.co.csk.vdm.toolbox.VDM.jdk.*;
// ***** VDMTOOLS END Name=imports

public class Stack implements EvaluatePP {

// ***** VDMTOOLS START Name=vdmComp KEEP=NO
    static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
// ***** VDMTOOLS END Name=vdmComp

// ***** VDMTOOLS START Name=stack KEEP=NO

```

```

    private volatile Vector stack = null;
// ***** VDMTOOLS END Name=stack

// ***** VDMTOOLS START Name=sentinel KEEP=NO
    volatile Sentinel sentinel;
// ***** VDMTOOLS END Name=sentinel

// ***** VDMTOOLS START Name=StackSentinel KEEP=NO
    class StackSentinel extends Sentinel {

        public final int Pop = 0;

        public final int Top = 1;

        public final int Push = 2;

        public final int Reset = 3;

        public final int Stack_u_u0_u = 4;

        public final int Stack_u_u1_ub_un_uStackItem = 5;

        public final int Stack_u_u1_ub_us_un_uStackItem = 6;

        public final int nr_functions = 7;

        public StackSentinel () throws CGException {}

        public StackSentinel (EvaluatePP instance)
            throws CGException {
            init(nr_functions, instance);
        }

    }
// ***** VDMTOOLS END Name=StackSentinel
;

// ***** VDMTOOLS START Name=evaluatePP KEEP=NO
    public Boolean evaluatePP (int fnr)
        throws CGException {
        return new Boolean(true);
    }
// ***** VDMTOOLS END Name=evaluatePP

```

```

// ***** VDMTOOLS START Name=setSentinel KEEP=NO
public void setSentinel () {
    try {
        sentinel = new StackSentinel(this);
    }
    catch (CGException e) {
        System.out.println(e.getMessage());
    }
}
// ***** VDMTOOLS END Name=setSentinel

// ***** VDMTOOLS START Name=Reset KEEP=NO
public void Reset () throws CGException {

    sentinel.entering(((StackSentinel) sentinel).Reset);
    try {
        stack = (Vector)
            UTIL.ConvertToList(UTIL.clone(new Vector()));
    }
    finally {
        sentinel.leaving(((StackSentinel) sentinel).Reset);
    }
}
// ***** VDMTOOLS END Name=Reset

// ***** VDMTOOLS START Name=Pop KEEP=NO
public Integer Pop () throws CGException {

    if (!this.pre_Pop().booleanValue())
        UTIL.RunTime("Run-Time Error:Precondition failure in Pop");
    sentinel.entering(((StackSentinel) sentinel).Pop);
    try {

        Integer res = UTIL.NumberToInt(stack.get(0));
        {

            stack = (Vector)
                UTIL.ConvertToList(UTIL.clone(new Vector(stack.subList(1, stack.size()))));
            return res;
        }
    }
    finally {

```

```

        sentinel.leaving(((StackSentinel) sentinel).Pop);
    }
}
// ***** VDMTOOLS END Name=Pop

// ***** VDMTOOLS START Name=pre_Pop KEEP=NO
public Boolean pre_Pop () throws CGException {
    return new Boolean(!UTIL.equals(stack, new Vector()));
}
// ***** VDMTOOLS END Name=pre_Pop

// ***** VDMTOOLS START Name=Push KEEP=NO
public Vector Push (final Integer val) throws CGException {

    sentinel.entering(((StackSentinel) sentinel).Push);
    try {

        Vector rhs_2 = null;
        Vector var1_3 = null;
        var1_3 = new Vector();
        var1_3.add(val);
        rhs_2 = (Vector) var1_3.clone();
        rhs_2.addAll(stack);
        stack = (Vector) UTIL.ConvertToList(UTIL.clone(rhs_2));
        return stack;
    }
    finally {
        sentinel.leaving(((StackSentinel) sentinel).Push);
    }
}
// ***** VDMTOOLS END Name=Push

// ***** VDMTOOLS START Name=Top KEEP=NO
public Integer Top () throws CGException {

    if (!this.pre_Top().booleanValue())
        UTIL.RunTime("Run-Time Error:Precondition failure in Top");
    sentinel.entering(((StackSentinel) sentinel).Top);
    try {
        return UTIL.NumberToInt(stack.get(0));
    }
    finally {
        sentinel.leaving(((StackSentinel) sentinel).Top);
    }
}

```

```

    }
}
// ***** VDMTOOLS END Name=Top

// ***** VDMTOOLS START Name=pre_Top KEEP=NO
public Boolean pre_Top () throws CGException {
    return new Boolean(!UTIL.equals(stack, new Vector()));
}
// ***** VDMTOOLS END Name=pre_Top

// ***** VDMTOOLS START Name=Stack KEEP=NO
public Stack () throws CGException {

    try {
        setSentinel();
    }
    catch (Exception e){

        e.printStackTrace(System.out);
        System.out.println(e.getMessage());
    }
    try {

        stack = (Vector) UTIL.ConvertToList(UTIL.clone(new Vector()));
        setSentinel();
    }
    catch (Throwable e) {
        System.out.println(e.getMessage());
    }
}
// ***** VDMTOOLS END Name=Stack

// ***** VDMTOOLS START Name=Stack KEEP=NO
public Stack (final Integer val) throws CGException {

    try {
        setSentinel();
    }
    catch (Exception e){

        e.printStackTrace(System.out);
        System.out.println(e.getMessage());
    }
}

```

```

        try {

            {

                Vector rhs_2 = null;
                rhs_2 = new Vector();
                rhs_2.add(val);
                stack = (Vector) UTIL.ConvertToList(UTIL.clone(rhs_2));
            }
            setSentinel();
        }
        catch (Throwable e) {
            System.out.println(e.getMessage());
        }
    }
}
// ***** VDMTOOLS END Name=Stack

// ***** VDMTOOLS START Name=Stack KEEP=NO
public Stack (final Vector init) throws CGException {

    try {
        setSentinel();
    }
    catch (Exception e){

        e.printStackTrace(System.out);
        System.out.println(e.getMessage());
    }
    try {

        stack = (Vector) UTIL.ConvertToList(UTIL.clone(init));
        setSentinel();
    }
    catch (Throwable e) {
        System.out.println(e.getMessage());
    }
}
// ***** VDMTOOLS END Name=Stack

}
;

```