Department of Computer Science
University of Manchester
Manchester M13 9PL, England

Technical Report Series
UMCS–92–12–4

Bob Fields

A guide to reading VDM specifications

A guide to reading VDM specifications*

Bob Fields

Department of Computer Science
University of Manchester
Oxford Rd., Manchester, U.K.
bob@cs.man.ac.uk

Abstract

The intention of this document is to give an overview of the syntax of the Specification Language of the Vienna Development Method (VDM) as an aid to those who are to some extent familiar with formal notations and have to read or review documents containing VDM. This document is **not** a tutorial introduction to or motivation for formal methods in general or VDM in particular; nor is it a formal description of what VDM is or means. Both these types of description can be found in the references.

*This work was supported both by the DTI Teaching Company Scheme.

Technical Reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `m1.cs.man.ac.uk` (130.88.13.4) in the directory `/pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, U.K.

1 General concepts

VDM ...

1. Is a *Formal Method*. *Formal* because specifications are written in a precise and unambiguous notation based on Predicate Logic and in which questions of a program's correctness can be reduced to mathematical statements. A *Method* because a framework is provided for progressing from abstract specifications to more concrete ones and ultimately to code. The framework is not described in this document, but more information can be found in the literature.

2. Is *Model-oriented* – operations of the system are specified in terms of models defined by abstract data types.

3. *Operations* and *functions* can be given an explicit (constructive) definition or an implicit specification in terms of pre- and post-conditions.

4. Is a name sometimes used to refer to the VDM specification language (rather than the VDM *method*). This language is also known as *Meta-IV* or VDM-SL. See Section 9 on the versions of VDM which are in existence.

Elements of a specification

A VDM specification essentially has four kinds of component.

Type equations introduce new type names as synonyms for type expressions. Types and composites may have *invariants* which restrict the set of values which are considered *legal*.

Composites or record types (tuples with labelled fields) are introduced using the syntax described in Section 4.1.

Functions are defined which take arguments and return results. The syntax of function definitions is given in Section 4.2.

Operations also may take arguments and return results and in addition have access to a 'state', which is a model for the global state of the system being modelled. The syntax for operations is given in Section 4.3.

Structure of the paper

The primitive types and type constructors which are available in VDM are described in the following sections as are the symbols used to construct the expressions which appear in the pre- and post-conditions of operations and functions and the bodies of functions.

Section 2 describes the syntax of the expressions which are used in function, operation and invariant definitions. Section 3 describes the basic types and type constructors used in type and composite declarations and the operators which are applicable to instances of these types. Section 4 shows how expressions and type-terms are used to make definitions and an example in Section 5 shows how specifications are constructed. The issue of definedness in the VDM logic is mentioned briefly in Section 7 and Sections 8 and 9 give references to VDM support tools and more complete VDM texts.

2 Expressions

2.1 Propositions

Propositions are terms which can denote truth values (i.e. **true** or **false**) and assuming P_1 and P_2 are propositions, can be formed in the following ways.

Expression	Read as	Meaning
$\neg P_1$	Not P_1	True if P_1 is false and is false if P_1 is true.
$P_1 \wedge P_2$	P_1 And P_2	True if both P_1 and P_2 are true and false if either P_1 or P_2 (or both) is false.
$P_1 \vee P_2$	P_1 Or P_2	True if either P_1 or P_2 (or both) are true and is false if both P_1 and P_2 are false.
$P_1 \Rightarrow P_2$	P_1 Implies P_2	Is the normal implication of logic and is defined as $\neg P_1 \vee P_2$.
$P_1 \Leftrightarrow P_2$	P_1 If-and-only-if P_2	True if either both P_1 and P_2 are true, or both are false (also read as 'P_1 is-equivalent-to P_2').

2.2 Typing judgements

The assertion 'a: X' says that some expression a is *of type* X. Such assertions are used most often as declarations (e.g. of the formal parameters in the header of an operation), but also appear in the binding part of quantifier or binder expressions. In common with typing constructs in many programming languages, a shortened form can be used for declaring several identifiers at once: 'a, b: T' is equivalent to 'a: T; b: T'. A similar syntax is used for parts of binding expressions introduced below.

2.3 Equality

Equality ($=$) can be tested between any two terms of the language *which denote values and are of the same type!* The equality test between two elements of different disjoint types is undefined, so for example, if $n: \mathbb{N}$ and $b: \mathbb{B}$, then '$n = b$' does not denote a value. Inequality ($\neq$) is defined in the obvious way in terms of equality and negation.

2.4 Predicate logic and binders

Variables may be used to range over values and binders such as quantifiers can be used to bind these variables. The notation $P[x]$ used below simply means that P is some arbitrary formula involving some (possibly zero!) *free occurrences* (i.e. not bound by any binder inside P) of the variable x.

Expression	Name	Meaning
$\forall x: T \cdot P[x]$	Universal Quantifier	For all elements x of the type T, some property P holds.
$\exists x: T \cdot P[x]$	Existential Quantifier	There is an x of type T such that the property P holds.
$\exists!\, x: T \cdot P[x]$	Unique Existential Quantifier	There is a *unique* x of type T such that P holds.
$\nexists x: T \cdot P[x]$	Not Exists	There is *no* element x of type T such that the property P holds.
$\iota\, x: T \cdot P[x]$	Iota-function	The value of 'the *unique* x of type T such that P holds. If there is more than one such x, then the expression fails to denote a value. Notice that this is closely related to the unique existential quantifier: $\iota\, x: T \cdot P[x]$ is defined if and only if $\exists!\, x: T \cdot P[x]$ is true.
$\lambda x: T \cdot P[x]$	Lambda Abstraction	Used for introducing *anonymous* functions. For example, the function application $(\lambda x: T \cdot P[x])t$ is equivalent to $P[t]$.

Expressions involving binders are considered equivalent up to the renaming of bound variables, so for instance:

$$\lambda x: T \cdot E[x] = \lambda y: T \cdot E[y]$$

and

$$\forall x: A \cdot \forall y: B \cdot P[x, y] = \forall y: A \cdot \forall x: B \cdot P[y, x]$$

In the examples of quantifiers above (i.e. $\forall$, $\exists$, etc., but not λ), all the variables are constrained to range over types, whereas what is often wanted is for the variable to range over some *set* of values. The effect of constraining the quantification to range over some set s – an element of type T-**set** – can be achieved by writing

$$\forall x: T \cdot x \in s \;\Rightarrow\; P[x]$$

(which is read as 'for all elements x of type T, if x is an element of set s, then $P[x]$') and the 'derived' shorthand form '$\forall x \in s \cdot P[x]$' is frequently used (where '$x \in s$' is read as 'element x is a member of set s – see Section 3.3). (Similarly $\exists x \in s \cdot P[x]$ cn be used as a shorthand form of $\exists x: T \cdot x \in S \wedge P[x]$.) Another useful derived form, applicable to all quantifiers, is

$$\forall x_1: T_1, x_2: T_2, \ldots, x_n: T_n \cdot P[x_1, x_2, \ldots, x_n]$$

which is an abbreviation for

$$\begin{array}{l} \forall x_1: T_1 \cdot \\ \quad \forall x_2: T_2 \cdot \\ \quad\quad \ldots \\ \quad\quad \forall x_n: T_n \cdot \\ \quad\quad\quad P[x_1, x_2, \ldots, x_n] \end{array}$$

2.5 Conditional expressions

A convenient construct is the conditional expression:

$$\textsf{if } c \textsf{ then } x \textsf{ else } y$$

which has the value x if c is true and the value y if c is false and is undefined in c is undefined.

2.6 Selection expressions

A piece of notation which can often remove the need to write a number of conditional expressions is the cases construct, where the *pattern expressions*, pat_i involve 'make functions' (see Section 4.1) or types and the expression E is an element of a union type (see Section 3.8).

$\mathsf{cases}\ E\ \mathsf{of}$
$pat_1 \rightarrow E_1$
$pat_2 \rightarrow E_2$
$\ldots$
$\mathsf{others}\ O$
end

The above formula has the value E_1 if E *matches* the pattern pat_1, E_2 if E *matches* pat_2 and so on. Because of this matching property, the cases construct effectively introduces a binding for variables occurring it the pat_i which "extends" through the corresponding E_i. See Section 5 for an example of how patterns and selections are used.

2.7 Abbreviations

Abbreviations can be introduced by the let mechanism:

$\mathsf{let}\ v = E_1\ \mathsf{in}$ E_2	$\mathsf{let}\ v \in E_1\ \mathsf{in}$ E_2

In the first example of the use of the let construct, the name v is simply introduced as an abbreviation for the expression E_1 and the entire let expression is equivalent to expression E_2 with every occurrence of v replaced by E_1. In the second example, however, E_1 denotes a *set* of values and v is used throughout E_2 to stand for an arbitrary element of this set. Note that the let construct is *deterministic*: any implementation of a specification involving the second type of let expression must *always* pick the same value for v for a given E_1, even though the specification doesn't say *which* value!

3 Types

3.1 Basic types

VDM has a number of 'built-in' basic types.

$\mathbb{B}$	Booleans	The two-element set of truth values $\{\mathsf{true}, \mathsf{false}\}$. Propositions and predicates, if they are well formed, are of type $\mathbb{B}$ and can be treated as data values like any other expression.
$\mathbb{N}$	Natural numbers	$\{0, 1, 2, \ldots\}$.
$\mathbb{N}_1$	Non-zero naturals	$\{1, 2, \ldots\}$.
$\mathbb{Z}$	Integers	$\{\ldots, -2, -1, 0, 1, 2 \ldots\}$.
$\mathbb{Q}$	Rational numbers	All $\frac{n}{m}$ where $m, n \in \mathbb{Z}$ and $m \neq 0$

A basic value, nil, is provided which can be used to represent certain sorts of 'error' values and is used to denote the absence of optional items (see Section 4.1). It is assumed that all the necessary arithmetic operators are defined on numbers in VDM (e.g. addition, multiplication, powers, roots, etc.)

3.2 Constants and enumerated types

A VDM specification may introduce a number of named constants or tokens, simply by writing them down. Enumerated types are defined by listing the elements of the enumeration, separated by the '|' symbol (which is also used for forming type unions – see Section 3.8), or by enclosing them in curly braces (using the same syntax as for set enumerations – see Section 3.3). For example, the following two types each contain three symbols.

$$\mathit{PopGroups} = \textsc{Stones} \mid \textsc{Beatles} \mid \textsc{Stranglers}$$

$$\mathit{MoreGroups} = \{\textsc{Fall}, \textsc{Pogues}, \textsc{Jam}\}$$

It is common to introduce types containing token-like objects without saying anything about the elements of the type. For instance, in a description of the abstract syntax of a programming language it would be useful to introduce types called *VarName* and *ProcName* for variable and procedure names, but it would not be reasonable to enumerate their elements, nor very instructive to give a definition of their lexical structure. Such types would typically be introduced by a note in the text (rather than in the formal part of the specification) to the effect that '*VarName* and *ProcName* are infinite, disjoint sets of structureless tokens'.

3.3 Finite sets

VDM provides the type constructor 'X-set' for constructing the type of finite sets of elements of type X (where X is some type expression containing primitive types, user-defined type names and type constructors). Assuming

$$e_1, e_2, e_3 \colon X \quad \text{and} \quad S_1, S_2 \colon X\text{-}\mathsf{set}$$

then the following operators are defined on sets.

$\{\}$	The empty set.	
$\{e_1, e_2, e_3\}$	Set enumeration	The set containing only the elements e_1, e_2 and e_3. (Similar to enumerated types above, but forming an expression, rather than a type.)
$\{i, ..., j\}$	Numeric range	For $i, j: \mathbb{Z}$, the set $\{x: \mathbb{Z} \mid i \leq x \leq j\}$.
$\{e: X \mid P[e]\}$	Set comprehension	Those e's in type X for which predicate P holds.
$\{e \in S \mid P[e]\}$	Set comprehension	Those e's in set S for which predicate P holds. Defined in terms of comprehension over types in a similar way as for other binders in §2.4.
$e_1 \in S_1$	e_1 is in S_1	The assertion that element e_1 is in the set S_1.
$e_1 \notin S_1$	e_1 is not in S_1	The assertion that element e_1 is not in the set S_1.
$S_1 \subset S_2$	S_1 is a proper subset of S_2	Every element of S_1 is contained in S_2 and S_1 and S_2 are not equal.
$S_1 \subseteq S_2$	S_1 is a subset of S_2	Every element of S_1 is contained in S_2 (S_1 and S_2 may be equal).
$S_1 \cap S_2$	S_1 intersect S_2	All the elements which are in *both* S_1 *and* S_2.
$S_1 \cup S_2$	S_1 union S_2,	All the elements which are in *either* S_1 *or* S_2.
$S_1 - S_2$	S_1 minus S_2	The set difference of S_1 and S_2, that is all the elements which are in S_1 but *not* in S_2.
$\mathsf{card}\, S_1$	Cardinality of S_1	The number of elements in S_1.
$\bigcup ss$	Distributed union of ss	If ss is a set of sets, then $\bigcup ss$ is the result 'unioning' together all of the sets in ss.
$\bigcap ss$		The intersection of all the sets in the set of sets ss.

Set Comprehensions Notice that two forms are given for set comprehension, both of which are *bounded*: in the first case the variable e is free only to range over the elements of a type and in the second, it ranges over the elements of a well-defined set. This restriction allows Russell-style paradoxes to be ruled out, so one cannot write expressions like '$\{x \mid x \notin x\}$' and expect to be able to prove anything about them! A useful derived, or shorthand form of set comprehension allows expressions of the form:

$$\{foo(x) \mid P[x]\}$$

where x ranges over elements of type A and *foo* is some function of type $A \to B$. The 'full' form of this expression is

$$\{y: B \mid \exists x: A \cdot P[x] \wedge foo(x) = y\}$$

When comprehensions of the abbreviated form are written, care must be taken to ensure that they really are abbreviations of the second form and that the typing information could be added if necessary.

3.4 Finite sequences

The sequence type constructor is provided to construct list types: X^*, denotes the type of finite sequences of elements of type X and X^+ denotes the type on non-empty sequences of X's (those with one or more element). A number of 'operators' are defined for sequences. Assuming

$$e: X \quad \text{and} \quad S_1, S_2: X^*$$

then the following expressions can be used to construct new sequences or manipulate existing ones.

$[]$	The empty sequence	
$[e_1, e_2, e_3]$	Sequence enumeration	The sequence containing e_1, e_2 and e_3
$\mathsf{cons}(e, S_1)$	Cons of e and S_1	The result of appending element e to the front of sequence S_1.
$\mathsf{len}\, S_1$	Length of S_1	The number of elements in S_1. (Zero if the list is empty.)
$\mathsf{hd}\, S_1$	Head of S_1	The first element of S_1. Undefined if S_1 is empty.
$\mathsf{tl}\, S_1$	Tail of S_1	A copy of S_1 with the head removed. Undefined if S_1 is empty.
$S_1 \frown S_2$	Concatenation	The result of joining S_1 and S_2 together, with S_1 at the front.
$\mathsf{inds}\, S_1$	Indices of S_1	If $\mathsf{len}\, S_1 = n$ then $\mathsf{inds}\, S_1$ is the set of numbers $\{1, \ldots, n\}$, or the empty set if S_1 is empty.
$\mathsf{elems}\, S_1$	Elements of S_1	A set containing exactly those elements which are in S_1.
$\mathsf{dconc}\, ss$	Distributed sequence concatenation	ss is a sequence of sequences and $\mathsf{dconc}\, ss$ is the result of concatenating them all together into one sequence.
$S_1[n]$	Indexing	The nth element of sequence S_1 if $n \in \mathsf{inds}\, S_1$, undefined otherwise.
$S_1[m, ..., n]$	Subsequencing	The sequence with elements of $[S_1[m], S_1[m+1], ...S_1[n]]$. from position m to position n. For this to be defined, $m \leq m$ and $n \leq \mathsf{len}\, S_1$.

Notice that there is no notation for sequence comprehension as there is for set comprehension. One might be tempted to think that some construct like

$$[x{:}\, T \mid P[x]]$$

could be used to mean a sequence of x's for which P holds true. This specifies which elements are in the sequence, but says nothing about their order. There are several ways in which the order may be defined, but they tend to be be a little cumbersome and none has gained widespread acceptance.

3.5 Cartesian products

The type of pairs of elements where the first element is of type A, and the second of type B is formed using the type constructor '$\times$' and denoted by '$A \times B$'. Two functions **fst** and **snd** are defined which respectively project out the first and second elements of such pairs. The pairing type constructor can be generalised to a 'tupleing' constructor to allow the definition of types like

$$T = T_1 \times T_2 \times T_3$$

Elements of pair or tuple types are formed simply by enclosing the components in angle brackets: $< a, b, c >$. When there are more than two elements in a tuple it does not really make much sense to introduce general operators like fst and snd to project out each component (see, however, Section 4.1). In order to get at the components, it is assumed that the cartesian product constructor is right associative, so that the type denoted by $A \times B \times C$ is equivalent to the one denoted by $A \times (B \times C)$, the expression $< a, b, c >$ is equal to $< a, < b, c >>$ and $\mathsf{fst}\,\mathsf{snd} < a, b, c > = b$ (and so on). This looks messy, but in practice the use of pattern matching (Section 2.6) and record types (Section 4.1) mean that this kind of selection from tuples is needed less frequently.

3.6 Finite maps

The type $A \xrightarrow{m} B$ denotes the type of finite mappings from type A to type B and $A \xleftrightarrow{m} B$ is the type of one-to-one maps from A to B. A mapping from type A to a type B is an association of some objects of type A to some objects of type B, and can be thought of as a finite set of ordered pairs of the form $< a, b >$, with the restriction that no two pairs in a mapping have the same first component. If a mapping, m, contains the pair $< x, y >$, then it is said that 'm maps x to y'. The restriction that the first tuple component be unique means that and a can only map to a *unique* b. A one-to-one mapping is one where the second components of the tuples are unique as well as the first.

If

$m_1, m_2 \colon A \xrightarrow{m} B;$
$a \colon A;\ b \colon B;$
$as \colon A\text{-}\mathsf{set};\ bs \colon B\text{-}\mathsf{set}$

then the following symbols are defined (some of which look identical to some of the operators applicable to sets).

$\{\}$	The empty map	
$m_1(a)$	m_1 applied to a	The object to which a is mapped by m. Undefined if a is not in the domain of m.
$\{a \mapsto b, a' \mapsto b'\}$	Map enumeration	A map which gives B when applied to a and b' when applied to a'.
$\{a \mapsto b \mid P[a, b]\}$	Map comprehension	A map m which gives b when applied to a if and only if $P[a, b]$.
$\mathsf{dom}\, m_1$	Domain of m_1	The set of elements to which the map can be applied (i.e. the set containing the first component of all the tuples in m_1).
$\mathsf{rng}\, m_1$	Range of m_1	The set of B's which are 'mapped to' by m_1 (i.e. the set containing the second component of all the tuples in m_1).
$m_1 \dagger m_2$	m_1 overwrite by m_2	'$(m_1 \dagger m_2)(a)$' can be defined as 'if $a \in \mathsf{dom}\, m_2$ then $m_2(a)$ else $m_1(a)$.
$as \lhd m_1$	Domain restriction	'$(as \lhd m_1)(a)$' is defined as 'if $a \in as$ then $m_1(a)$ else undefined'.
$as \ntriangleleft m_1$	Domain subtraction	'$(as \ntriangleleft m_1)(a)$' can be defined as 'if $a \in as$ then undefined else $m_1(a)$'.
$m_1 \rhd bs$	Range restriction	'$(as \rhd m_1)(a)$' can be defined as 'if $m(a) \in bs$ then $m_1(a)$ else undefined'.
$m_1 \ntriangleright bs$	Range subtraction	'$(as \ntriangleright m_1)(a)$' can be defined as 'if $m(a) \in bs$ then undefined else $m_1(a)$'.

Mappings are perhaps an unfamiliar data type, so here are a few simple examples of their use. Suppose $m_1 = \{a \mapsto b, c \mapsto d, e \mapsto d\}$ and $m_2 = \{e \mapsto y\}$. Then the domains and ranges of the two maps are given by:

$$\mathsf{dom}\, m_1 = \{a, c, e\} \qquad \mathsf{rng}\, m_1 = \{b, d\}$$
$$\mathsf{dom}\, m_2 = \{e\} \qquad \mathsf{rng}\, m_2 = \{y\}$$

The maps may be applied to any element in their domain so that $m_1(a) = b$ and $m_1(c) = d$, but $m_2(a)$ is undefined.

The two mappings may be combined using the map overwrite operator: $m_1 \dagger m_2 = \{a \mapsto b, c \mapsto d, e \mapsto y\}$. Since the domain of mapping m_1 includes that of m_2, $m_2 \dagger m_1 = m_1$. Portions of a mapping may be selected using the domain and range deletion and restriction operators.

$$\{a, c\} \lhd m_1 = \{a \mapsto b, c \mapsto d\} \qquad \{a, c\} \ntriangleleft m_1 = \{e \mapsto d\}$$
$$m_1 \rhd \{d\} = \{c \mapsto d, e \mapsto d\} \qquad m_1 \ntriangleright \{b, d\} = \{\}$$

3.7 Optional items

In cartesian product types (and records too), it is useful to be able to have some components which are optional. The syntax of optional items is a type surrounded by square brackets. For example, lists of natural numbers can be defined as:

$$LinkedList = \mathbb{N} \times [LinkedList]$$

If an optional object is 'missing' it is equal to the special value 'nil'.

3.8 Union types

Types can be formed by taking the union of all the elements in other types using the '|' type constructor. For example, in

$$T = T_1 \mid T_2 \mid T_3$$

T contains all the elements of T_1, all those of T_2 and all those of T_3. T is sometimes called a sum instead of a union and each of the T_i are known as summands. Typical uses of union types in real specifications will involve a function of type $T \to \mathit{Something}$ which will use a case expression to decide which of the T_i the argument is an element of and specify some appropriate computation in each case. In such cases it must be possible to tell which unique summand each element of T comes from – the T_i must therefore be *disjoint*.

4 Making definitions

A VDM specification consists of a collection of definitions of named types composite or record types, possibly restricted by invariants, functions and operations. The following subsections introduce the notation for making these definitions.

4.1 Type definitions

Type synonyms

Type equations provide a means by which names or synonyms can be introduced for type expressions. For example

$$\mathit{MyMap} = \mathit{Thing} \xrightarrow{m} \mathit{OtherThing}\text{-}\mathsf{set}$$

defines the name *MyMap* as a synonym for the type of mappings from *Thing*s to sets of *OtherThing*s.

Record types

It would be convenient if, instead of having to use cartesian products for all composite objects and having to mess around with all of these fsts and snds, there were some way of defining a composite object with more than two fields (and occasionally with less than two!) and with some way of referring 'directly' to the components. Record or composite types allow exactly this: they are a means whereby several items of data of different types can be combined and treated as a single unit. Composites are similar to 'structure' types in C or S3 or 'records' in Pascal. They are introduced by the following syntax

$$\begin{array}{lrcl} \mathit{Rec\text{-}name} :: & \mathit{fieldname}_1 & : & \mathit{type}_1 \\ & \mathit{fieldname}_2 & : & \mathit{type}_2 \\ & & \ldots & \end{array}$$

for example:

$$\begin{array}{lrcl} \mathit{Date} :: & \mathit{day} & : & \mathbb{N}_1 \\ & \mathit{month} & : & \mathit{Month} \\ & \mathit{year} & : & \mathbb{N}_1 \end{array}$$

where

$Month = \text{JAN} \mid \text{FEB} \mid \ldots \mid \text{DEC}$

For each record type which is introduced in a specification a number of functions are implicitly defined. Firstly a constructor or 'make' function which allows elements of the record type to be built from elements of the component types. For the *Date* example this function is called *mk-Date* and an example of its use is $mk\text{-}Date(8, \text{SEP}, 1991)$. In addition to the constructor, a number of 'selector functions' (also known as *projector* or *destructor* functions) are implicitly defined to allow elements of the record type to be split up into their component bits. The selectors for the *Date* type are *day*, *month* and *year* which project out the first, second and third components respectively. If $d = mk\text{-}Date(8, \text{SEP}, 1991)$, then $day(d) = 8$, $month(d) = \text{SEP}$ and $year(d) = 1991$.

It is often necessary to update just one component of a record object, leaving all the others the same. For example, incrementing the *day* field of the date d above could be specified as

$mk\text{-}Date(day(d) + 1, month(d), year(d))$

For large records with many fields, this becomes somewhat verbose, so a derived shorthand form is provided by the 'μ' operator. This operator allows just one component of the record to be changed by the '$\mu(object, fieldname \mapsto value)$' construct which produces a new data element which is a copy of *object*, but in which the named field has the given value. The above example is equivalent to

$\mu(d, day \mapsto day(d) + 1)$

Invariants and subtypes

It is often convenient to restrict one's attention to 'those elements, x, of type T for which some property P holds'. VDM allows this kind of *subtype* formation by the use of invariants. Recall the *Date* type defined in Section 4.1: this is clearly not a complete definition of the type of dates since it permits such nonsensical elements as $mk\text{-}Date(35, \text{AUG}, 1892)$ and $mk\text{-}Date(29, \text{FEB}, 1900)$. The invariant of the following type eliminates such 'garbage' elements – the illegal objects simply aren't members of the type, because they do not satisfy the invariant.

$$
\begin{array}{lrl}
Date_1 :: & day & : \mathbb{N}_1 \\
 & month & : Month \\
 & year & : \mathbb{N}_1
\end{array}
$$

$$
\begin{array}{l}
\textsf{inv if } month \in \{\text{APR}, \text{JUN}, \text{NOV}, \text{SEP}\} \\
\quad \textsf{then } day \leq 30 \\
\quad \textsf{else if } month = \text{FEB} \\
\qquad \textsf{then if } isLeapYear(year) \\
\qquad\quad \textsf{then } day \leq 29 \\
\qquad\quad \textsf{else } day \leq 28 \\
\qquad \textsf{else } day \leq 31
\end{array}
$$

where *isLeapYear* is an appropriate boolean-valued function.

4.2 Function definitions

Now we are beginning to arrive at a point where actual specifications of programs can be written. For example the function quoted in the previous section can be defined as follows

$$isLeapYear : \mathbb{N}_1 \to \mathbb{B}$$

$$\begin{aligned} isLeapYear(y) \;\triangleq\; & \\ & ((y \;\mathsf{mod}\; 4 = 0) \wedge \neg (y \;\mathsf{mod}\; 100 = 0)) \vee \\ & y \;\mathsf{mod}\; 400 = 0 \end{aligned}$$

That is, a year is a leap year if either it is not a century and it is divisible by 4, or it is divisible by 400.

The general syntax of functions is

$$functionname : P_1 \times P_2 \times \ldots \to R$$

$$functionname(param_1, param_2, \ldots) \;\triangleq\; body$$

$$\mathsf{pre}\; precondition$$
$$\mathsf{post}\; postcondition$$

The first line of the definition gives the name and type, or *signature* of the function, so *isLeapYear* takes a single argument of type $\mathbb{N}_1$ and returns a boolean. A function with the signature

$$Thing \times \mathbb{N}^* \to \mathbb{N}\text{-}\mathsf{set}$$

takes two arguments – a *Thing* and a sequence of naturals – and returns a result which is a set of natural numbers.

The second line of the definition begins by giving the name and formal parameters of the function, followed by the symbol '$\triangleq$' which is read as 'is defined as'.

The pre and postconditions are boolean-valued expressions: the precondition places restrictions on the arguments to which the function may be applied[1]. The *body* is a definition of the function, which may be an expression, or it may be simply a variable, the possible values of which are restricted by the postcondition, a predicate (boolean valued expression) which states what must be true of the result value.

It is not generally useful to provide *both* a body and a postcondition – a function is defined either explicitly with a body, or implicitly with a postcondition. An exceptional circumstance is where the body is simply a variable, and the postcondition restricts the values permitted for this variable.

Occasionally it is convenient for ease of typesetting and for readability to use the 'operation syntax' (see the next section) to define functions. They will look like operations, but will, of course, have no access to a *state*. Because of the issue of under-specification versus non-determinism, which is explained briefly below, it is sometimes very significant whether something is an operation or a function and the accompanying text must make this clear.

Under specification

Functions may be 'under-specified' (sometimes called 'loosely specified'), in that the specification may permit several different return values (e.g. by the use of a post-condition), but they are, by definition, *deterministic*, in the sense that any implementation of the function must always return the same result for a particular argument, so that the important property that equality is reflexive (*viz* $x = x$) is maintained.

[1] If a function is applied to arguments for which the precondition is not satisfied, then the function is free return absolutely any result at all!

As an illustration of this consider the following function which *deterministically* picks an *arbitrary* element from a set.

$$pick : Thing\text{-}\mathsf{set} \to Thing$$

$$pick(s) \quad \triangleq \quad e$$

$$\mathsf{pre}\ s \neq \{\,\}$$
$$\mathsf{post}\ e \in s$$

It cannot *a priori* be determined which element is denoted by $pick(ss)$ (unless ss has only one element), but since $pick$ is a function, one can be certain that for the same ss, the result of $pick(ss)$ is always the same, so that if $ss = ss'$, then it must always be the case that $pick(ss) = pick(ss')$. Operations (and implementations of them), on the other hand need not be deterministic in this way.

Formal parameters

In the above example, $isLeapYear$, the parameter is simply given as an identifier y. It is, however, legal to use a 'pattern' expression, of the form '$mk\text{-}T(a, b, \ldots)$' as a parameter. For example

$$isLeapMonth : Date \to \mathbb{B}$$

$$isLeapMonth(mk\text{-}Date(d, m, y)) \quad \triangleq$$
$$\quad isLeapYear(y) \land m = \textsc{Feb}$$

which could be rewritten as either of:

$$isLeapMonth : Date \to \mathbb{B}$$

$$isLeapMonth(dd) \quad \triangleq$$
$$\quad isLeapYear(year(dd)) \land$$
$$\quad month(dd) = \textsc{Feb}$$

$$isLeapMonth : Date \to \mathbb{B}$$

$$isLeapMonth(dd) \quad \triangleq$$
$$\quad \mathsf{let}\ dd = mk\text{-}Date(d, m, y)\ \mathsf{in}$$
$$\quad isLeapYear(y) \land m = \textsc{Feb}$$

4.3 Operation definitions

The syntax of an operation specification is illustrated by the template:

$$op\ (p_1 \colon P_1, p_2 \colon P_2, \ldots)\ r \colon R$$
$$\mathsf{ext\ rd}\ rs \ :\ RS$$
$$\quad \mathsf{wr}\ ws \ :\ WS$$
$$\mathsf{pre}\ precondition$$
$$\mathsf{post}\ postcondition$$

The first line, or header, contains the name of the operation and the names and types of the formal parameters and the (optional) name and type of the result(s). Formal parameters to operations can be defined as patterns rather than just variables, in much the same way as for functions.

Following the operation header is a definition of the **ext**ernal 'state' of the operation, which can be thought of as the 'global machine state' of the system being specified, or a part thereof.

The state can either be declared in the operation as a single entity, like 's: *StateType*', or as a number of separate parts. The operation is given either read-only (**rd**) or read-write (**wr**) access to each 'state component' individually. If an operation is given read-only access to a state component, s, then this piece of state is referred to simply as 's' throughout the operation's definition. If, on the other hand, the operation is given read-write access to s, then the name '$\overleftarrow{s}$' is used to refer to the state immediately prior to the operation being executed (the 'old' state') and the name 's' is used to refer to the final state after execution (the 'new' state). In the precondition of an operation s is used to denote the state before execution, since it makes no sense to mention the final state in a precondition.

The rest of the specification contains two sections: the precondition and the postcondition. The precondition is a predicate on the arguments and initial state of the operation and indicates to which states and arguments the operation is applicable. The postcondition is a predicate on the arguments, initial and final states and the result variables, which specifies what must be true *after* the operation has been executed.

If an operation were applied to a state which does not satisfy the precondition, the situation is *not* invalid, but the specification does not guarantee *anything* about the final state or result: they are completely undefined. This means that an implementation could do anything to the state and return any result, and still be considered a 'correct' implementation of the specification. It is a good idea, therefore to avoid applying operations outside their preconditions!

An example of an operation which takes an arguments and returns no result, but has write access to a state is given below. Notice that a precondition is necessary to ensure that the operation is not applied on the 29th of February in a leap year.

$$
\begin{array}{l}
\mathit{TimeWarp}\ (n\colon \mathbb{N}) \\
\textsf{ext wr}\ dt\ :\ \mathit{Date} \\
\textsf{pre}\ \mathit{isLeapYear}(\mathit{year}(dt))\ \Rightarrow\ \neg\,(\mathit{month}(dt) = \text{FEB} \wedge \mathit{day}(dt) = 29) \\
\textsf{post}\ \mathit{day}(dt) = \mathit{day}(\overleftarrow{dt}) \wedge \\
\qquad \mathit{month}(dt) = \mathit{month}(dt) \wedge \\
\qquad \mathit{year}(dt) = \mathit{year}(\overleftarrow{dt}) + 1
\end{array}
$$

Sometimes an operation's precondition is been omitted. This means an implicit precondition of **true**: the operation is applicable to all states and inputs.

5 Example: a simple language

This section contains an example specification of a very simple system which should illustrate many of the aspects introduced so far.

The example is the definition of a simple language of terms. Terms are either variables, *Var*, or binary terms *BinTerm*.

$$\mathit{Term} = \mathit{Var} \mid \mathit{BinTerm}$$

It is assumed that *Var* is an infinite set of token-like objects, which do not need further definition. *BinTerms* have three fields: left and right operands and an operator (of which there are only two different sorts: multiplication and addition).

$$
\begin{array}{lrcl}
\mathit{BinTerm} :: & \mathit{left} & : & \mathit{Term} \\
& \mathit{op} & : & \mathit{Operator} \\
& \mathit{right} & : & \mathit{Term}
\end{array}
$$

$Operator = \{*, +\}$

A function which will turn out to be useful on terms, is *allVars*, which takes an term as an argument and returns the set of all variables which occur as subterms within the term.

$allVars : Term \rightarrow Var\text{-}\mathsf{set}$

$$
\begin{aligned}
allVars(e) \;\; \triangleq \;\; & \mathsf{cases}\ e\ \mathsf{of} \\
& mk\text{-}BinTerm(l, o, r) \rightarrow allVars(l) \cup allVars(r) \\
& \mathsf{others}\ \{e\} \\
& \mathsf{end}
\end{aligned}
$$

The final stage of this example is to define an interpreter for terms which can evaluate an term in the context of some *variable bindings*: the type *Env* records the correspondence between variables and numeric values.

$Env = Var \xrightarrow{m} \mathbb{N}$

The top-level interpreter is an operation, *eval*, which takes an term and returns a natural number. *eval* has read-only access to a *Env* whose purpose is to represent the "machine state" during the execution of an term. The precondition prevents terms from being evaluated if they contain variables which do not exist in the *Env*.

$$
\begin{aligned}
& eval\ (e{:}\ Term)\ val{:}\ \mathbb{N} \\
& \mathsf{ext\ rd}\ env\ :\ Env \\
& \mathsf{pre}\ allVars(e) \subseteq \mathsf{dom}\ env \\
& \mathsf{post\ cases}\ e\ \mathsf{of} \\
& \quad mk\text{-}BinTerm(l, o, r) \rightarrow \exists lv, rv{:}\ \mathbb{N} \cdot \\
& \qquad\qquad post\text{-}eval(l, env, lv) \wedge \\
& \qquad\qquad post\text{-}eval(r, env, rv) \wedge \\
& \qquad\qquad \mathsf{cases}\ o\ \mathsf{of} \\
& \qquad\qquad\ * \ \rightarrow\ lv * rv \\
& \qquad\qquad\ + \ \rightarrow\ lv + rv \\
& \qquad\qquad \mathsf{end} \\
& \quad \mathsf{others}\ env(e) \\
& \quad \mathsf{end}
\end{aligned}
$$

The above example, while very simple, illustrates some of the features of specifications of programming language manipulation systems (such as compilers and interpreters for *real* languages). The idea of directly modelling the state of a system (in this case by *Env*) is a very general technique which is fundamental to the VDM approach and can be used very widely.

6 Example: a text editor

An example of a slightly different application is a simple text editor which allows the manipulation of texts which consist of a sequence of lines, each of which is a sequence of characters.

$Text = Line^{+}$

$Line = Character^{+}$

Note that empty texts and empty lines within a text are not catered for – while being unrealistic, this simplifies the presentation.

The internal state of a text editor contains a text object and two numeric fields to record the coordinates of the editing cursor.

$$\begin{aligned} \mathit{EditorState} :: \; & \mathit{contents} : \mathit{Text} \\ & \mathit{x\_cursor} : \mathbb{N}_1 \\ & \mathit{y\_cursor} : \mathbb{N}_1 \end{aligned}$$

$$\begin{aligned} &\mathsf{inv}\ \mathit{y\_cursor} \leq \mathsf{len}\ \mathit{contents} \wedge \\ &\quad \mathit{x\_cursor} \leq \mathsf{len}\ \mathit{contents}[\mathit{y\_cursor}] \end{aligned}$$

The invariant on the editor state ensures that the cursor always points to a real character in the text, rather than to a line off the end of the text or off the end of a line.

An operation which this editor can perform is to scroll itself a certain number of lines.

$$\begin{aligned} &\mathit{scroll}\ (n : \mathbb{Z}) \\ &\mathsf{ext\ wr}\ e\ :\ \mathit{EditorState} \\ &\mathsf{pre}\ 0 < n + \mathit{y\_cursor}(e) \wedge n + \mathit{y\_cursor}(e) < \mathsf{len}\ \mathit{contents}(e) \\ &\mathsf{post\ let}\ e = \mathit{mk\text{-}EditorState}(xc, yc, \mathit{conts})\ \mathsf{in} \\ &\qquad \mathsf{if\ len}\ \mathit{conts}[n + yc] < xc \\ &\qquad \mathsf{then}\ \mathit{mk\text{-}EditorState}(\mathsf{len}\ \mathit{conts}[n + yc], n + yc, \mathit{conts}) \\ &\qquad \mathsf{else}\ \mathit{mk\text{-}EditorState}(xc, n + yc, \mathit{conts}) \end{aligned}$$

The precondition only permits attempts to scroll to lines which exist, and the effect of the operation is to move the cursor up or down the given amount. If the new line is shorter than the original one, then the cursor is positioned at the end of it, otherwise the original x-position is retained.

7 Treatment of definedness

The logic underlying VDM is called the Logic of Partial Functions (LPF) and differs from ordinary classical logic in that it permits reasoning about non-denoting or undefined terms. For example, terms like $\frac{1}{0}$ or $(-3)!$ cannot be said to denote a value. Because of its treatment of undefinedness, LPF is able to model several important concepts in software development, such as non-termination of programs and non-implementability of specifications.

Although LPF allows reasoning about undefined terms, it does not treat undefinedness explicitly (there is no third 'truth' value such as 'bottom', to represent non-termination), so when all terms are defined, reasoning is similar to that in classical logic with the difference that the 'law of the excluded middle' (i.e. everything is true or false) does not hold. LPF is defined in detail in [BCJ84] and the issue of logics to support software specification in general is discussed in [CJ90] and [DLO86, Owe85].

8 Support

This section describes some of the support packages which exist for VDM, but is meant to be an introduction to what is possible, rather than a comprehensive review. Tools exist for VDM specification and development on two levels: clerical support, for instance typesetting macro packages and *semantic* support to assist with typechecking, theorem proving and so on.

Typesetting

Two sets of LaTeX macro packages for formatting VDM specifications are worthy of mention. The first set was written by Mario Wolczko at Manchester University and has been used to typeset a number of books and papers, for example [Jon90, JJLM91] and this paper. These macros are called `vdm.sty` or `bsivdm.sty`.

The second macro package is called `vdmsl.sty` and is closer to the BSI standard. `vdmsl.sty` was written recently by Jan-Bert Oostenenk of the Hoger Informatica Onderwijs, Hogenschool Enschede, The Netherlands and is maintained by Graeme Parkin of The National Physical Laboratory.

Semantic support

The *mural* system [FEG92, JJLM91] has two main parts: a set of VDM support tools which maintain a database of specifications and an interactive theorem proving assistant. Specifications are generated using structure editors and similar tools and attempts cam be made using the proof assistant to show conjectures about the correctness and consistency of the specification. Certain conjectures (proof obligations about the internal consistency on the specification) are generated automatically by the tools for subsequent discharge by the user.

The *SpecBox* system provides a wide range of checking and bookkeeping facilities relating to the consistency of specifications written in the ASCII syntax of the BSI standard language. Specifications must be syntactically correct, functions must be defined and used with the same number of arguments throughout a specification and so on. A certain amount of simple type-checking is performed and SpecBox is capable of generation output which is acceptable as input to the mural system. In addition, SpecBox produces various cross references and indices of usage and definition.

9 Further reading

9.1 Dialects of VDM

Over the years, a number of dialects of VDM have been developed at different times, by different groups, or for different purposes. All are conceptually more or less the same and differ mostly in their concrete syntax.

Meta-IV

One of the earliest versions was known as *Meta-IV* and is described and applied by the papers in the collection [BJ78]. This notation was mainly used for defining the syntax and semantics of programming languages and is in some respects tuned to that application area though it is by no means restricted to it.

Bush, Cottam & Latham

A variant of VDM is used in [LBC90] as part of an undergraduate-level course on imperative programming (in Pascal) via formal specification. In one or two places the specification language has been modified to make it look a little more like the Pascal 'equivalent' (for example, composite types are introduced with the **record** keyword).

HP-SL

The Hewlett-Packard Specification Language [Bea91] is a VDM-like specification language developed at HP Labs in Bristol. The stated aim of the HP-SL work was to produce a specification language which was small and regular, while embodying enough flexibility and power to allow specification in a mixture if different styles. Some of the principal differences between HP-SL and other VDM languages are addition of features for modularisation and polymorphic definitions, and the level of tool support provided (syntax and type checking tools have been built for HP-SL).

Standardisation

The VDM specification language is currently undergoing standardisation by the British Standards Institute and the International Standards Organisation. See, for example, [L$^+$90], for the current state of the work on VDM's semantics. A number of books have been published which which use notations similar to the emerging standard. [AI91] and [Daw91] use the BSI/ISO notation. The BSI standard defines an 'ascii' syntax which can be printed and displayed on devices with restricted character sets.

Jones [Jon90], the main introductory VDM text, describes a (close) forerunner of the standard syntax; this paper essentially uses the same language as [Jon90].

9.2 Ongoing development

Numerous papers have been published which discuss VDM-related issues and research work in more depth. The logic on which VDM (and other formal methods) is based is discussed in [CJ90, BCJ84]. [Jon86] addresses the problems of relating specifications to real programs. Numerous projects have undertaken to extend or modify the VDM notation as an attempt to remedy some of its deficiencies (yes, it has some!) and to provide capabilities use in specific application areas. Constructs for specifying concurrent systems and parallelism have been described by [Mid89, Stø90, WD88].

The issue of modularity and structuring of specifications is an important one for any specification language and method which is to be adopted for large-scale projects. Several proposals for adding modular constructs to VDM have been made, for example in the VVSL dialect [Mid89] and in dialects closer to the BSI standard notation (see, for example the thesis [Fit91]). [FJ90] presents a "challenge problem" in writing modular specifications using a VDM modular mechanism.

[JS90] contains a number of case-study specifications and developments in VDM. The textbook [Rol92] gives an introduction to the philosophy of formal methods in general and a VDM in particular, with an emphasis on methodological aspects. Relationships between VDM and object-oriented design are stressed and examples and case studies are given where specifications are implemented as programs in Modula-2 and Ada.

Acknowledgements

Several people at ICL and Manchester University have made helpful comments and suggestions for improving this guide. Thanks in particular to Brian Monahan of Manchester University for useful suggestions and discussions on the content, presentation and technical accuracy of this document and to Stephanie Wilson of Queen Mary and Westfield College for pointing numerous

inconsistencies and omissions in earlier versions. I shall, of course, take all the credit myself for any remaining errors or omissions.

References

[AI91] D. Andrews and D. Ince. *Practical Formal Methods with VDM.* McGraw-Hill, 1991.

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[Bea91] S. Bear. An overview of HP-SL. In S. Prehn and W.J. Toetenel, editors, *VDM'91 – Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 1991. Vol.1: Conference Contributions.* Lecture Notes in Computer Science *551.*, pages 571–587. Springer-Verlag, 1991.

[BJ78] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-language*, volume 61 of *LNCS.* Springer-Verlag, 1978.

[CJ90] J.H. Cheng and C.B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J.C.P. Woodcock, editors, *Procedings of the Third Refinement Workshop.* Springer-Verlag, 1990.

[Daw91] J. Dawes. *The VDM-SL Reference Guide.* Pitman, 1991. ISBN 0-273-03151-1.

[DLO86] O.J. Dahl, D.F. Langmyhr, and O. Owe. Preliminary report on the specification and programming language ABEL. Research Report 106, Department of Informatics, University of Oslo, 1986.

[FEG92] B. Fields and M. Elvang-Gøransson. A VDM case study in *mural. IEEE Transactions on Software Engineering*, April 1992. to appear.

[Fit91] J.S. Fitzgerald. *Modularity in Model-Oriented Formal Specifications and its Interaction with Formal Reasoning.* PhD thesis, Department of Computer Science, University of Manchester, 1991.

[FJ90] J. S. Fitzgerald and C.B. Jones. Modularizing the formal description of a database system. Technical Report UMCS-90-1-1, Manchester University, Department of Computer Science, January 1990.

[JJLM91] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. mural: *A Formal Development Support System.* Springer-Verlag, 1991.

[Jon86] C.B. Jones. Program specification and verification in VDM. Technical Report UMCS-86-10-5, Manchester University, Department of Computer Science, November 1986.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 2nd edition, 1990.

[JS90] C.B. Jones and R.C.F. Shaw, editors. *Case Studies in Systematic Software Development.* Prentice-Hall International, 1990.

[L$^+$90] P.G. Larsen et al. The dynamic semantics of the BSI/VDM specification language. Technical report, The Institute of Applied Computer Science (IFAD), Odense, Denmark, August 1990.

[LBC90] J.T. Latham, V.J. Bush, and I.D. Cottam. *The Programming Process: An Introduction using VDM and Pascal.* International Computer Science Series. Addison Wesley, 1990.

[Mid89] C.A. Middelburg. *Syntax and Semantics of VVSL – A Language for Structured VDM Specifications.* PhD thesis, University of Amsterdam, 1989.

[Owe85] O. Owe. An approach to program reasoning based on a first order logic for partial functions. Research Report 89, Department of Informatics, University of Oslo, 1985.

[Rol92] F.D. Rolland. *Programming with VDM.* Macmillan, 1992.

[Stø90] K. Stølen. *Development of parallel programs on shared data structures.* PhD thesis, Department of Computer Science, University on Manchester, 1990.

[WD88] J.C.P. Woodcock and B. Dickinson. Using VDM with rely and guarantee conditions: Experiences from a real project. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM '88* VDM – The Way Ahead. *LNCS 328*, pages 434–458. Springer-Verlag, 1988.