Typesetting BSI VDM with LaTeX

Mario Wolczko
Dept. of Computer Science
The University
Manchester M13 9PL
U.K.
mario@cs.man.ac.uk, ...!uknet!man.cs!mario

09 June 1992
Version 3.01

Contents

1 Overview

This document describes a style option, `vdm`, for use with LaTeX. The purpose of `vdm` is to make the typesetting of VDM specifications easy. Other goals are:

- To enable users of `vdm` to communicate their specifications to others, possibly in a variety of concrete syntaxes, without having to change their source files
- To enable a user of `vdm` to concentrate on his[1] specifications, and ignore the detailed layout as much as possible. A side effect of this is that the effort required to improve layout is concentrated in one place, within the `vdm` macros.

(This version of the `vdm` style option uses the BSI concrete syntax. Any document prepared using earlier versions is still accepted, but the way it is typeset will match more closely the BSI standard concrete syntax. There are also a few additional commands (summarised at the end). Note that this is *not* a complete style file for all of BSI VDM.)

But enough evangelising. Let's get to the the real meat. This document is broken up into the following sections:

- General points about using `vdm`
- Typesetting formulas
- How to typeset data types
- How to typeset functions
- How to typeset operations
- How to typeset proofs
- How to tailor/extend the system for your own application.

You should definitely read the first two sections—then you'll know roughly what you're in for, and whether you want to continue. The remaining sections can be read as and when you need them.

In keeping with the best traditions of TeX documentation, paragraphs that contain material that is not essential for novices, but vital if you want to parameterise or extend the system, are in smaller type, like this one.

Just to give a preliminary example, here is some output from `vdm`, and the corresponding input:

[1] Read 'his/her' for every occurence of 'his'.

$$dec : Oop\text{-}\mathbf{set} \times Oop \xrightarrow{m} Object \to Oop \xrightarrow{m} Object$$

$$\begin{array}{l} dec(ptrs, om) \triangleq \\ \quad \textbf{if } ptrs = \{\} \\ \quad \textbf{then } om \\ \quad \textbf{else let } gone = \{p \in ptrs \mid RC(om(p)) = 1\} \textbf{ in} \\ \qquad \textbf{let } om' = gone \lhd om \textbf{ in} \\ \qquad \textbf{let } om'' = om' \dagger \{p \mapsto \mu(om'(p), RC \mapsto RC - 1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid p \in ptrs - gone\} \textbf{ in} \\ \qquad dec(\bigcup\{\textbf{elems}\, BODY(om(p)) \mid p \in gone\}, om'') \end{array}$$

$$\begin{array}{l} DESTROYPTR\ (Obj, Ptr{:}\ Oop) \\ \textbf{ext wr } OM\ :\ Oop \xrightarrow{m} Basic\_Object \\ \textbf{pre } ptr \in \textbf{elems}\, BODY(om(obj)) \\ \textbf{post } om = \overleftarrow{om} \dagger \{obj \mapsto \mu(om(obj), BODY \mapsto BODY - \{ptr\})\} \end{array}$$

```
\begin{vdm}
\begin{fn}{dec}{ptrs,om} \\
\signature{
      \setof{Oop} \x \mapof{Oop}{Object} \to \mapof{Oop}{Object}
}
\If ptrs = \emptyset
\Then om
\Else   \Let gone = \set{p \in ptrs | RC(om(p)) = 1} \In
        \Let om' = gone \dsub om \In
        \Let om'' = om' \owr
                \map{p \mapsto \chg{om'(p)}{RC}{RC\minus 1}
                        | p \in ptrs \diff gone} \In
        dec(\Union\set{\elems{BODY(om(p))} | p \in gone}, om'')
\Fi
\end{fn}

\begin{op}[DESTROYPTR]
\args{ Obj, Ptr : Oop }
\ext{ \Wr OM : \mapof{Oop}{Basic_Object} }
\pre{ ptr \in  \elems{BODY(om(obj))} }
\post{ om = ~{om} \owr \map{ obj \mapsto
                \chg{om(obj)}{BODY}{BODY \diff \set{ptr}}}}
\end{op}
\end{vdm}
```

2 Using vdm—General Points

To get at vdm, include vdm as a document style option, e.g.:

```
\documentstyle[12pt,vdm]{report}
```

To the best of my knowledge, the use of `vdm` does not conflict with any of the other document styles, except when something has been redefined. An attempt will be made to document all such redefinitions.

Once `vdm` has been included, you can then use the `vdm` environment. For example,

```
\begin{vdm}
   ....
\end{vdm}
```

All specification material should be placed within the `vdm` environment. The use of `vdm` only affects text within the `vdm` environment, except for the following global changes (which are only relevant when in math or display math mode):

1. The mathcodes of a...z and A...Z have been changed. In plain English, this means that when you type letters in math mode the inter-letter spacing may be different than it would be had you not included `vdm` as an option.[2] This is because LaTeX math mode is usually tuned for single letter identifiers, as used by mathematicians for millenia. However, you and I both know that most meaningful identifiers have more than one letter in them, so `vdm` provides better spacing for them. As an example, if you type `$identifier$`, LaTeX would normally print $identifier$, whereas the use of `vdm` will yield $identifier$.

   If you really want to use the 'normal' inter-letter spacing, say `\defaultMathcodes`.

2. Underscore gives you an underscore, and not a subscript. If you want a subscript use `@`, e.g., x_0 is typed `x@0`, or use TeX's `\sb` macro. An `@` is still an @ when not in math mode. Occasionally you may find that an @ in math mode *doesn't* give you a subscript (particularly when used with moving arguments). Should this happen, you are advised to use TeX's `\sb` macro, e.g., `$x\sb{0}$`.

   If you don't use underscores much, and you want to use `_` for subscripts, you can say `\underscoreon` (and `\underscoreoff` to make it revert to its usual meaning in `vdm`).

3. `-` typesets a hyphen, and not a minus sign. VDM specifications usually contain a lot more *long-identifiers* than subtractions, so on the whole this alteration should save effort. If you really want to do a single subtraction sign, use `\minus`. If you find the default is inappropriate, you can revert to the original behaviour using `\mathminus`; `\textminus` is the inverse. Example: `a-b \ne\mathminus a-b` gives $a\text{-}b \neq a - b$.

[2] This is not the case if you are using PSLaTeX, as that does not distinguish text italic from math italic.

4. `|` gives you a |, and not a |. Do you see the difference? No? The former goes between things, e.g., $\{x \mid p(x)\}$, while the latter is a delimiter, e.g., $|x|$. In VDM, most people use the former more than the latter, so again this seems reasonable. If you really want a | (the second kind), say `\vert`.

5. In TEX and LATEX `~` has always been a tie (a space between words at which the line is never broken). Well in `vdm` it isn't. `~x` will give you a $\overleftarrow{x}$. For long identifiers, such as $\overleftarrow{long}$, say `~{long}`. *Note that this only applies in math mode; elsewhere a ~ is still a tie.*

6. In math mode, the double quote character `''` is actually a macro. Placing text between pairs of double quotes causes that text to be set in the normal text font. For example, `$x="a variable"$` gives you $x =$ a variable.

   If you want to change the font used for text placed between quotes, redefine the command `\mathTextFont`. By default it is defined to be `\rm` (`\mathrm` for the New Font Selection Scheme).

7. The following macros have been altered in a non-trivial way: `\forall`, `\exists` (see later).

When you typeset some VDM within the `vdm` environment, by default it is set in from the left margin by an amount equal to `\parindent`, the indentation at the beginning of each paragraph. If you want to change this, change the value of `\VDMindent`, e.g.:

```
\setlength{\VDMindent}{0cm}
```

will make your specs come out flush left. This document has been typeset with `\VDMindent` equal to $3 \times$ `\parindent`.

Similarly, the right hand margin is controlled by a parameter called `\VDMrindent`. By default it is also set to `\parindent`.

You can have a particular line spacing in force within the `vdm` environment. The spacing within a `vdm` environment is dictated by the `\VDMbaselineskip` command. Note that this is *not* a length, but a command. By default it expands to `\baselineskip` so that the line spacing is that of the surrounding text, whatever size that may be. To make it smaller, you may want to say

```
\renewcommand{\VDMbaselineskip}{0.8\baselineskip}
```

for example.

3 Typesetting formulas

Most of the text you enter within `vdm` environments will be in TEX's math mode, but VDM does its best to conceal this fact from you, so that you should rarely, if ever, have to type a dollar sign. However, several new features have been

provided for the typesetting of logical formulas. Firstly, operators with sensible names have been provided: use `\Iff`, `\Implies`, `\Or`, `\And` and `\Not` for the operators $\Leftrightarrow, \Rightarrow, \vee, \wedge$ and $\neg$. (To retain compatibility with a previous version, `\iff`, `\implies`, `\and` and `\neg` are still provided, but `\or` is not.)

A major change has come in the area of quantified expressions. In VDM, they have very well-defined forms, so the LaTeX sequences `\forall` and `\exists` have been re-defined to take arguments. For example, to get

$$\exists x \in S \cdot p(x)$$

type

```
\exists{x \in S}{p(x)}
```

Note the separating dot that was put in automatically. If you want one of these dots by itself, you can have one by saying `\suchthat`.

In addition, two new quantifiers, `\unique` and `\nexists`, have been added:

```
\unique{x \in S}{p(x)}
\nexists{x \in S}{p(x)}
```

$$\exists! \, x \in S \cdot p(x)$$

$$\nexists \, x \in S \cdot p(x)$$

Additionally, to complement `\unique`, there is `\uniqueval`. This is the so-called "iota-function" that returns the unique value, if there is one:

```
\uniqueval{x \in S}{p(x)}
```

$$\iota \, x \in S \cdot p(x)$$

If you want to use the old versions of `\forall` and `\exists` they are available under the pseudonyms of `\Forall` and `\Exists`.

If you find that the body of the quantified expression is too long to fit comfortably on a line, there are *-forms of the above commands that place the body of the quantified expression on a new line, slightly indented. For example,

$$\exists x \in S \cdot \\ \quad p(x) \wedge q(x) \vee \neg p(x) \;\Rightarrow\; r(x) \wedge S(x)$$

can be obtained with

```
\exists*{x \in S}{p(x) \And q(x) \Or \Not p(x)
                  \Implies r(x) \And S(x)}
```

If you need "Strachey" brackets, e.g., $M[\![e]\!]$, place the material to appear within the brackets within `\term{ ... }`, thus: `$M\term{e}$`.

A special control sequence, `\const`, is available for constants. To get, for example, YES | NO, type `\const{Yes}|\const{No}`.

If you don't like the font that constants are set in, you can change them by redefining the command `\constantFont`. By default it expands to `\sc`.

3.1 The `formula` Environment

Occasionally you may want a formula on its own, between paragraphs of text, say. Normally, the provided environments and commands suffice, but sometimes

they don't. If you need an odd equation to stand on its own, use the `formula` environment:

```
\begin{formula}
x = 10
\Or  \forall{i \in \Nat}{i \ne 10 \Implies i \ne x}
\end{formula}
```

The `formula` environment is similar to displayed math mode, except: formulas are indented by `\VDMindent`, not `\mathindent`, and line breaks can be made using `\\`. Also, within the `formula` environment everything appears flush left, as opposed to being centred.

3.2 Constructions

A particularly nice feature of `vdm` is that you can typeset multi-line constructions such as those in the earlier example without having to worry about, say, lining up "thens" and "elses" with "ifs". In the following definitions, whenever you see the term ⟨math-mode-expression⟩, you should type an expression as if in math mode, but you needn't put dollar signs in. All of the constructions described below can be used where a ⟨math-mode-expression⟩ is required. Each construction is shown by example; the output on the left results from the input on the right. Also note that each macro name begins with an upper-case letter. TEX and LATEX frequently use the lower-case variants for completely unrelated things. Naturally, chaos will ensue if you mix the names up.

Typesetting an **if** is done using `\If` ⟨math-mode-expression⟩`\Then` ⟨math-mode-expression⟩`\Else` ⟨math-mode-expression⟩`\Fi`.

if $x \in S$
then $S - x$
else $\{\}$

```
\If x\in S
\Then S \diff x
\Else \emptyset
\Fi
\end{verbatim}
```

If you nest `\If`s then you must enclose inner `\If`s within braces:

if ...
then if ...
  then ...
  else ...
else

```
\If ...
\Then{
        \If ...
        \Then ...
        \Else ...
        \Fi
}\Else
\Fi
```

You are advised to place the extra braces exactly as above; don't let extraneous spaces intervene between the keywords and the braces.

The `\If` macro always starts a new line for the **then** and **else** parts. If you want TEX to try to choose line breaks, use `\SIf` instead:

```
\SIf a=b
\Then c=d+e
\Else p=q+r+s+t+u
\Fi
```

if $a = b$ **then** $c = d + e$
else $p = q + r + s + t + u$

let...**in** constructions are done in a similar way: `\Let` ⟨math-mode-expression⟩ `\In` ⟨math-mode-expression⟩, and `\SLet` ⟨math-mode-expression⟩ `\In` ⟨math-mode-expression⟩.

```
\Let x=f(y,z) \In
g(x)+h(x)
```

let $x = f(y, z)$ **in**
$g(x) + h(x)$

```
\SLet x=f(y,z) \In{x^2}
```

let $x = f(y, z)$ **in** x^2

Notice that `\SLet` takes a second argument, which is part of the same 'paragraph', where `\Let` does not.

The typesetting of a **cases** clause is more complicated. It takes the form:

`\Cases{` ⟨math-mode-expression⟩`}`
from-⟨math-mode-expression⟩`&` to-⟨math-mode-expression⟩`\\`
from-⟨math-mode-expression⟩`&` to-⟨math-mode-expression⟩`\\`
...
`\Otherwise{` ⟨math-mode-expression⟩`}`
`\Endcases`

The `\Otherwise` field is optional. This construction follows a general pattern that is common in `vdm` input: lists of things are separated by `\\`s, and subfields are separated by `&`s or `:`s.

In reality, there is another, optional argument, after the `\Endcases`. If you were to try typesetting something like

```
(... var = \Cases ...
           \Endcases)
```

you'd find the closing right parenthesis in an unexpected place (on the same line as the =, in fact). To get text to the right of the `\Endcases` you can place an optional argument within brackets after it:

```
(... var = \Cases ...
           \Endcases[)]
```

Admittedly, this looks a little strange, but it does work.

Here is an example of `\Cases` in action:

cases *select*(x) **of**
nil $\rightarrow \{\,\}$
mk-Lst$(hd, tl) \rightarrow \{hd\} \cup$ **elems** tl
others x
end

```
\Cases{ select(x) }
\nil            & \emptyset \\
mk-Lst(hd,tl)  & \set{hd} \union \elems{tl}
\Otherwise{ x }
\Endcases
```

Note the \\ is a *separator* and not a *terminator*—you don't need one after the last item. Also, the `\Otherwise` can appear anywhere between the `\Cases{}` and the `\Endcases`, but it will always be typeset last.

Some people prefer the selectors to appear lined up on the left, some on the right. If you want them to appear on the left, say `\leftCases`; if you want them on the right, say `\rightCases`. The scope of the `\leftCases` and `\rightCases` commands is the current group. By default, you get `\rightCases`.

3.2.1 The `formbox` Environment

Occasionally you might find that you want to put a line break in a place that can't handle \\. For example, if you have a `\Cases` command and the rhs of a particular case is too big, you can't use \\ to break the line directly, as it will be interpreted as the separator between cases. Then you must the `formbox` environment. It is similar to the `formula` environment in that you can put all sorts of things in it, but it can be used within other constructions, unlike the `formula` environment, which can only be used at the outermost level.

This example should convey the general idea:

```
\Cases{ f(x) }
mk-Very_long_constructor(foo,bar) &
    {\begin{formbox}
    long_predicate_with(foo) \\
      \And long_predicate_with(bar)
    \end{formbox}}
...
```

cases $f(x)$ **of**
$mk\text{-}Very\_long\_constructor(foo, bar) \rightarrow long\_predicate\_with(foo)$
$\qquad\qquad \wedge\ long\_predicate\_with(bar)$
$\qquad\qquad \ldots$
end

Note the extras braces around the `formbox`; these are required to "hide" the \\ from the `\Cases`.

3.3 Other General Points about Formulas

\\ will[3] *always* start a new line. Sometimes this is done in addition to some other function (as in the `\Cases` macro, where it delimits a particular case), but

[3] For 'will' read 'should'.

you should be able to use \\ almost anywhere to force a line break. Indeed, sooner or later you'll want to typeset a long formula and TEX will not be able to break the line sensibly, or will choose an unpleasant break. In this case you'll have to use \\.

Frequently you need to indent things within multi-line formulas. To help you do this, a command is provided which breaks a line, and indents the next line by an amount which you can supply (in units of `ems`). The `\T` command takes a single argument that controls how much the next line will be indented:

```
a \And b \T2
\Implies b \And a \T1
\Or d \And e
```

$$a \wedge b \\ \quad \Rightarrow\ b \wedge a \\ \vee\ d \wedge e$$

Along similar lines is the `\R` command. This does a line break, like \\, but then pushes the formula on the next line as far to the right as it can:

```
(a \And b \Implies b \And a) \R
\Or d \And e
```

$$(a \wedge b\ \Rightarrow\ b \wedge a) \\ \vee\ d \wedge e$$

Beware: it may end up pushing it further to the right than you expected! This is A Bug, and Will Not Be Fixed, so you'll have to work around it.

The `\If`, `\Let`, etc., constructions are all unusual in that it's impossible to typeset something sensibly to the right of them. For example, if you try

```
\exists{x \in S}{
   \If x=0 \Then S=Q \Else S=P \Fi}
\Or S=\emptyset
```

then you'll get

$$\exists x \in S \cdot \textbf{if } x=0 \textbf{ then } S=Q \textbf{ else } S=P \\ \vee\ S=\{\}$$

which is unlikely to be what you wanted.

You should also remember that where `vdm` wants a ⟨math-mode-expression⟩, TEX will be placed in math mode. This is usually the right thing to do, but occasionally you might want a natural language comment to appear there. In this case you'll have to insert an `\mbox` or a `\parbox` depending on whether your comment might span one or more lines:

```
\If \mbox{the condition is true}
\Then \mbox{do the true part}
\Else "do the false part"
\Fi
```

if the condition is true
then do the true part
else do the false part

The else-part illustrates how quotes can be used an an abbreviation for `\mbox{...}` within math mode.

Finally, all the constructions above will not break at a page boundary. This means that you're in big trouble if you want to typeset a three-page `\Cases`. The only statement I can make to mitigate this is: you shouldn't have expressions

that complicated in the first place—who do you expect to read them? Remember: "truth is beauty", so if your formulas are not beautiful, then chances are they're not true either.

4 Typesetting data types

The following table lists the primitive types and values available:

$\{0, 1, \ldots\}$	$\mathbb{N}$	`\Nat`
$\{1, 2, \ldots\}$	$\mathbb{N}_1$	`\Natone`,`\Nati`
$\{\ldots, -1, 0, 1, \ldots\}$	$\mathbb{Z}$	`\Int`
Rationals	$\mathbb{Q}$	`\Rat`
Real numbers	$\mathbb{R}$	`\Real`
$\{\mathbf{true}, \mathbf{false}\}$	$\mathbb{B}$	`\Bool`
Truth	**true**	`\true`,`\True`
Falsehood	**false**	`\false`,`\False`
Nil	**nil**	`\nil`

If you need a new keyword, you can create one easily. For example, if your favourite brand of logic has "maybe" as a value, you can say

```
\makeNewKeyword{\maybe}{maybe}
```

and henceforth `\maybe` is a valid control sequence that produces the text **maybe**. The text of the second argument to `\makeNewKeyword` can be anything; it doesn't have to match your control sequence name.

If you don't like the font that keywords are set in, you can change it by redefining the command `\keywordFontBeginSequence`. By default it expands to `\sf`.

The following type-related commands are provided:

Output	Input	
x-**set**	`\setof{x}`	set type constructor
$\{a, b, c\}$	`\set{a,b,c}`	set enumeration
$\{\,\}$	`\emptyset`	the empty set
x^*	`\seqof{x}`	seq. type constructor
$[a, b, a, c]$	`\seq{a,b,a,c}`	seq. enumeration
$[\,]$	`\emptyseq`	the empty sequence
$x \xrightarrow{m} y$	`\mapof{x}{y}`	map type constructor
$x \xmapsto{m} y$	`\mapinto{x}{y}`	one-one map type
$\{p \mapsto x\}$	`\map{p\mapsto x}`	map enumeration
$\{\,\}$	`\emptymap`	the empty map

Here are the relevant operators:

$\in$	`\in`	$\dagger$	`\owr`	$\frown$	`\sconc`
$\notin$	`\notin`	$\lhd$	`\dres`	**len** l	`\len{l}`
$\subset$	`\subset`	$\rhd$	`\rres`	**hd** l	`\hd{l}`
$\subseteq$	`\subseteq`	$\unlhd$	`\dsub`	**tl** l	`\tl{l}`
$\cap$	`\inter`,`\intersection`;	$\unrhd$	`\rsub`	**elems** l	`\elems{l}`
$\bigcap$	`\Inter`,`\Intersection`;	**dom** m	`\dom{m}`	**inds** l	`\inds{l}`
$\cup$	`\union`	**rng** m	`\rng{m}`	**dconc** l	`\Conc{l}`
$\bigcup$	`\Union`	**min** s	`\Min{s}`	**cons**(h, t)	`\cons{h,t}`
$-$	`\diff`,`\difference`;	**max** s	`\Max{s}`		
card s	`\card{s}`				

If you invent a new monadic keyword operator (like **dom** , etc.), then you can have `vdm` define for you a control sequence which switches font, and puts the right spacing in. For example,

```
\newMonadicOperator{\inv}{inv}
```

will define the `\inv` control sequence to print **inv**. Henceforth you can say, e.g., `\inv{Foo}`. All such sequences take one argument (they are monadic, after all).

You can define a new type using `\type{`type-name`}{`type`}`:

```
\type{Complex}{\Real\x \Real}
```

$Complex = \mathbb{R} \times \mathbb{R}$

Composites types can be typeset using the `composite` environment:

```
\begin{composite}{Datec}
  day :\set{1,\ldots,366}, \\
  year:\set{1583,\ldots,2599}
\end{composite}
```

compose *Datec* **of**
day : $\{1, \ldots, 366\}$,
year : $\{1583, \ldots, 2599\}$
end

There is also a `composite*` environment (and an equivalent `\scompose` control sequence) that places the entire composite type on a single line:

```
\begin{composite*}{Celsius}
  \Real
\end{composite*}
\scompose{Celsius}{\Real}
```

compose *Celsius* **of** $\mathbb{R}$ **end**

compose *Celsius* **of** $\mathbb{R}$ **end**

'Records' can be defined using the `record` environment:

`\begin{record}{`record-type-name`}`
field-name `:` field-type `\\`
...
`\end{record}`

The colons are used as sub-field separators.

```
\begin{record}{PERSON}
  NM : \seqof{Char} \\
  FEM : \Bool
\end{record}
```

PERSON :: *NM* : *Char**
FEM : $\mathbb{B}$

If the definition is short, you may prefer to use a short form:

```
\defrecord{PERSON}{
  NM : \seqof{Char} \\
  FEM : \Bool
}
```

Some people prefer the field names to appear lined up on the left, some on the right. If you want them to appear on the left, say `\leftRecord`; if you want them on the right, say `\rightRecord`. The scope of the `\leftRecord` and `\rightRecord` commands are the current group. By default, you get `\rightRecord`.

Updating fields of composites using the μ-function can be specified using `\chg`:

```
\chg{p}{FEM}{\Not man(q)}
```

$\mu(p, FEM \mapsto \neg\, man(q))$

Notice that the μ, parentheses, comma and $\mapsto$ were inserted automatically.

5 How to Typeset Functions

Typesetting λ-expressions is easy:

```
\LambdaFn{x,y}{x^2+y^2}
```

$\lambda x, y \cdot x^2 + y^2$

As with `\forall`, `\exists` and `\unique`, `\LamdbaFn` has a *-form that places the body of the function below and to the right:

```
\LambdaFn*{x,y,z}{
  (x^2+y^2+z^2)^{\frac12}}
```

$\lambda x, y, z \cdot$
$(x^2 + y^2 + z^2)^{\frac{1}{2}}$

There is also a `fn` (function) environment for defining named functions. It has the following structure:

`\begin{fn}`{name-of-function}`{` argument-list `}`
`\signature{`signature-of-function`}`
⟨optional precondition⟩
⟨optional postcondition⟩
body of function (a ⟨math-mode-expression⟩)
`\end{fn}`

See the third page for an example. The `\signature` is optional and can be placed anywhere within the body—it will always be typeset before the body. Useful macros within the `\signature` are: `\x` and `\to`, which yield $\times$ and $\rightarrow$. Note that you can also enter functions defined implicitly with pre- and post-conditions; see the next section on how to enter them.

All of the material in the section on formulas is relevant within the body of the function.

If you frequently intersperse your function definitions with text (and you should), you can save some typing by using the `vdmfn` environment.

`\begin{vdmfn}` ... `\end{vdmfn}` is equivalent to `\begin{vdm}\begin{fn}` ... `\end{fn}\end{vdm}`.

The `fn` environment also has a *-form that does not insert parentheses around the argument list. For example:

$MP[\![p]\!]\rho\sigma \triangleq \ldots$

```
\begin{fn*}{MP}{
  \term{p}\rho\sigma}
...
\end{fn*}
```

If you require the $\triangleq$ symbol by itself, then you can get it by saying `\DEF`.

5.1 Invariants

To typeset an invariant on a composite object, use the following structure:

$D :: day : Day$
$\quad year : Year$
where
$inv\text{-}D(mk\text{-}D(d, y)) \triangleq$
$\quad is\text{-}leapyr(y) \lor d \leq 365$

```
\begin{record}{D}
  day : Day \\
  year : Year
\end{record}
\where
\begin{fn}{inv-D}{mk-D(d,y)}
  is-leapyr(y) \Or d \le 365
\end{fn}
```

6 How to Typeset Operations

Operations are typeset within the `op` environment. The general structure is:

`\begin{op}[`⟨name-of-operation⟩`]`
`\args{`⟨list-of-arguments⟩`}`
`\res{`⟨result(s)⟩`}`
`\ext{`⟨list-of-externals⟩`}`
⟨pre-condition⟩
⟨post-condition⟩
`\end{op}`

The order of the various parts within the `op` environment is not important; they will always be printed in a canonical style (see page 3 for an example).

Any of `\args`, `\res`, `\ext`, ⟨pre-condition⟩ or ⟨post-condition⟩ may be omitted. `\begin{vdmop}` is an abbreviation for `\begin{vdm}\begin{op}`; `\end{vdmop}` is an abbreviation for `\end{op}\end{vdm}`.

The ⟨name-of-operation⟩ can be any one-line expression; it is typeset in math mode. An alternative way of specifying the name of the operation is to omit the optional argument (within `[]`), and use `\opname{`⟨name-of-operation⟩`}`, anywhere within the body of the `op` environment.

The ⟨list-of-arguments⟩ is a ⟨math-mode-expression⟩ that can span multiple lines; force a newline with `\\`. If present it is placed within parentheses.

The ⟨result(s)⟩ is also any ⟨math-mode-expression⟩. It is typeset to the right of any arguments.

The ⟨list-of-externals⟩ takes the following form:

```
\ext{
  ⟨optional \Rd or \Wr⟩ ⟨external-name(s)⟩ : ⟨external-types⟩ \\
  ⟨optional \Rd or \Wr⟩ ⟨external-name(s)⟩ : ⟨external-types⟩ \\
...
}
```

Alternatively, if the list of externals is long (say, more than five lines) the `externals` environmment can be used:

```
\begin{externals}
  ⟨optional \Rd or \Wr⟩ ⟨external-name(s)⟩ : ⟨external-types⟩ \\
  ⟨optional \Rd or \Wr⟩ ⟨external-name(s)⟩ : ⟨external-types⟩ \\
...
\end{externals}
```

Some people prefer the externals identifiers to appear lined up on the left, some on the right. If you want them to appear on the left, say `\leftExternals`; if you want them on the right, say `\rightExternals`. The scope of the `\leftExternals` and `\rightExternals` commands are the current group. By default, you get `\leftExternals`.

The ⟨pre-condition⟩ and ⟨post-condition⟩ take similar forms:

```
\pre{⟨math-mode-expression⟩}
```

or

```
\begin{precond}
⟨math-mode-expression⟩
\end{precond}
```

and

```
\post{⟨math-mode-expression⟩}
```

or

```
\begin{postcond}
⟨math-mode-expression⟩
\end{postcond}
```

Use the `\begin`...`\end` style if the ⟨math-mode-expression⟩ is longer than a few lines. All of the constructs mentioned in the section on formulas can be used within pre- and post-conditions.

7 Proofs

Here's an example of typesetting proofs in the "natural deduction" style.

from $E_1 \vee E_2$

1 **from** E_1

infer $E_2 \vee E_1$ $\vee$-I(h1)

2 **from** E_2

infer $E_2 \vee E_1$ $\vee$-I(h2)

infer $E_2 \vee E_1$ $\vee$-E(h,1,2)

```
\begin{proof}
    \From E@1 \Or E@2                          \\
1       \From E@1                              \\
        \Infer E@2 \Or E@1  \by $\vee$-I(h1)   \\
2       \From E@2                              \\
        \Infer E@2 \Or E@1  \by $\vee$-I(h2)   \\
    \Infer E@2 \Or E@1      \by $\vee$-E(h,1,2) \\
\end{proof}
```

Proofs are embedded within the `proof` environment. (A proof does not have to be within a `vdm` environment.) Each line of the proof ends with `\\`. Lines that begin a subproof have `\From` after the equation number. Lines that end a subproof have `\Infer` after the equation number. Other lines have `\&` after the equation number (see next example). If you don't need an equation number, just omit it, but you must have one of either `\From`, `\Infer` or `\&` on each proof line. If you want to include a justification of a particular proof line at the right hand end of the line, type it after a `\by`. `\by` is optional; you needn't include it if you don't need a justification.

Points worth bearing in mind:

- You are automatically placed in math mode after the `\From`, `\Infer` or `\&`; the math mode ends at the next `\by` or `\\`.
- You *cannot* break a line in the middle simply by using `\\` before `\by`; you must use separate proof lines to split a formula.
- You are within a `tabbing` environment within a proof, so you can use all the usual `tabbing` commands (`\=`, `\>`, etc.) to line things up across proof lines. Note that you will explicitly have to enter math mode again after any of these commands though.

Here's another example:

from $\forall x \in X \cdot E(x); s \in X$

1	$\neg \exists x \in X \cdot \neg E(x)$	$\forall$-defn(h)
2	$\neg \neg E(s/x)$	$\neg \exists$-E(1,h)

infer $E(s/x)$

```
\begin{proof}
  \From \forall{x\in X}{E(x); s\in X}                          \\
1 \&    \Not\exists{x\in X}{\Not E(x)} \by $\Forall$-defn(h)\\
2 \&    \Not\Not E(s/x)             \by $\Not\Exists$-E(1,h)\\
  \Infer E(s/x)                                                \\
\end{proof}
```

The amount of space used by the proof number can be changed by changing the length `\ProofNumberWidth`. The distance from the left margin to the proof number is dictated by `\ProofIndent`.

8 Customising the Style

Some people are never satisfied. We all know that it's true. In order to cater for those who aren't satisfied with the output from `vdm`, some attempt has been made to allow a limited degree of customisation by the user. In particular, you can alter some of the internal spacing chosen by `vdm`, and even have your own macros called at chosen places within `vdm`'s macros. Naturally, you are not advised to try this unless you feel you have some idea of what you want, and what you are doing. In this section we list the things that you can change, in order of increasing difficulty.

8.1 Changing the Spacing

In several places, essentially arbitrary spacings have been chosen by the author. The dimensions of these spaces are given by *rubber lengths.*[4] If you want to change any of them, use LaTeX's `\setlength` or `\addtolength` commands. For example,

```
\setlength{\postHeaderSkip}{13.33pt plus 2pt minus 1pt}
```

The `plus` and `minus` parts of a length let you say how much that length can grow or shrink by. For example, `12pt plus 2pt minus 1pt` means that the length will be in the range 11–14pt, with 12pt as its "natural" length.

The spaces in question all appear around `vdm` items such as operations, and in between major parts of such items. The names of the lengths should convey

[4] See the LaTeX book for an explanation of rubber lengths

where they apply. The following table lists all the lengths, and their default settings. Note that an `ex` is about the height of an "x" in the current font, and an `em` is about the width of an "M" in the current font.

Length	*Default size*	*Used within*
`\preOperationSkip`	2ex + 0.5ex − 0.2ex	`op` env
`\postOperationSkip`	2ex + 0.5ex − 0.2ex	
`\postHeaderSkip`	.5ex + .2ex − .2ex	
`\postExternalsSkip`	.5ex + .2ex − .2ex	
`\postPreConditionSkip`	.5ex + .2ex − .2ex	
`\preFunctionSkip`	2ex + .5ex − .2ex	`fn` env
`\postFunctionSkip`	2ex + .5ex − .2ex	
`\betweenSignatureAndBodySkip`	1.2ex + .3ex − .2ex	
`\betweenFunctionAndPreSkip`	1.2ex + .3ex − .2ex	
`\preTypeSkip`	1.2ex + .5ex − .3ex	`type` command
`\postTypeSkip`	1.2ex + .5ex − .3ex	
`\preCompositeSkip`	1.2ex + .5ex − .3ex	`composite` env
`\postCompositeSkip`	1.2ex + .5ex − .3ex	
`\preRecordSkip`	.75ex + .3ex − .2ex	`record` env
`\postRecordSkip`	.75ex + .3ex − .2ex	
`\preFormulaSkip`	1.2ex + .5ex − .3ex	`formula` env
`\postFormulaSkip`	1.2ex + .5ex − .3ex	
`\preProofSkip`	.75ex + .3ex − .2ex	`proof` env
`\postProofSkip`	.75ex + .3ex − .2ex	

8.2 Controlling Line and Paragraph Breaks

TEX uses the notion of *penalties* to decide where line and page breaks go. Various values of penalty are used at places within `vdm` to control breaks. To fully understand how to choose breaks, read *The TEXbook*. However, put simply, penalties are whole numbers in the range −10000 to 10000. A value of 10000 means "never break here," and a value of −10000 means "always break here." Values in between penalise or encourage breaking proportionally, so that, e.g., a value of −500 encourages a break, but by no means forces it. A value of zero is neutral.

To assign to a penalty `\p`, write `\p=1000`, for example. The table below list the penalties used by `vdm`, and their default values.

Penalty Name	Default Value	Where Used
\preOperationPenalty	0	`op` env
\preExternalPenalty	2000	
\prePreConditionPenalty	800	
\prePostConditionPenalty	500	
\postOperationPenalty	-500	
\preFunctionPenalty	0	`fn` env
\betweenSignatureAndBodyPenalty	500	
\betweenFunctionAndPrePenalty	1000	
\postFunctionPenalty	-500	
\preRecordPenalty	0	`record` env
\postRecordPenalty	-100	
\preProofPenalty	-100	`proof` env
\postProofPenalty	0	
\preFormulaPenalty	-100	`formula` env
\postFormulaPenalty	0	

8.3 Unforeseen Changes

It is a truism that no matter how good the designer of a piece of software is, he can never foresee all of its uses. In this case, the author is quite certain that people will use `vdm` for all sorts of things apart from typesetting VDM specifications. To cater for those who find that `vdm` does almost, but not quite, what they want, a number of *hooks* have been left in place. These hooks are macros, which at the moment do little or nothing, but which can be redefined by users to change the basic operation of `vdm`(see the `vdmindex` style for one such use). Needless to say, anyone wishing to redefine a hook should already be competent in the ways of LaTeX at least, and probably TeX as well. Rather than trying to explain what the hooks do, and where they do it, the user should look through the commented version of `vdm` (usually stored as `vdm.doc`) and figure it out for himself. Below are listed all the provided hooks, their default definitions, and where they are used.

Name of hook	*Default definition*
`op` environment	
`\preOperationHook`	`\penalty\preOperationPenalty␣`
`\betweenHeaderAndExternalsHook`	`\penalty\preExternalPenalty␣`
`\betweenExternalsAndPreConditionHook`	`\penalty\prePreConditionPenalty␣`
`\betweenPreAndPostConditionHook`	`\penalty\prePostConditionPenalty␣`
`\postOperationHook`	`\penalty\postOperationPenalty␣`
`fn` environment	
`\preFunctionHook`	`\penalty\preFunctionPenalty␣`
`\betweenSignatureAndBodyHook`	`\penalty\betweenSignatureAndBodyPenalty␣`
`\betweenFunctionAndPreHook`	`\vskip-\lastskip␣` `\vskip\betweenFunctionAndPreSkip` `\penalty\betweenSignatureAndBodyPenalty␣`
`\postFunctionHook`	`\penalty\postFunctionPenalty␣`
`record` environment	
`\preRecordHook`	`\penalty\preRecordPenalty␣`
`\postRecordHook`	`\penalty\postRecordPenalty␣`
`proof` environment	
`\preProofHook`	`\penalty\preProofPenalty␣`
`\postProofHook`	`\penalty\postProofPenalty␣`
`formula` environment	
`\preFormulaHook`	`\penalty\preFormulaPenalty␣`
`\postFormulaHook`	`\penalty\postFormulaPenalty␣`

9 Installing the vdm files

Place the file `vdm.sty` in your standard directory for LaTeX style files (your system administrator will know where this is). If you have the AMS fonts, change the appropriate line in `vdm.sty` (see instructions at the head of the file).

10 New vdm commands (introduced for the BSI version)

- There is a new keyword, `\rem`.
- Operations can also have an *error condition* part, typeset after the post-condition. The error condition is placed in an `errcond` environment. An alternative short form, `\err`, is also available, which works in the same way as `\pre` and `\post`.

  In support of this new part, there is a hook, `\betweenPostAndErrConditionHook`, defined to be `\penalty \preErrConditionPenalty` (the default penalty is 500). The preceding white space is defined by `\preErrConditionSkip` (default .5ex + .2ex − .2ex).
- `\Others` is an alias for `\Otherwise`.
- Sequents are supported using the `sequent` command, thus:

$$A \vdash B$$
$$Truth, Beauty, \textit{eq-intr} \vdash Truth = Beauty$$

```
\sequent{A}{B}
\sequent*{Truth,Beauty,eq-intr}
  {Truth=Beauty}
```

- Optional items can be typeset using `\Opt`, thus:

$$[fred]$$

```
\Opt{fred}
```

- There are two new monadic operators, `\abs` and `\merge`.
- A non-empty sequence type can be defined using `\neseqof`, thus:

$$\mathbb{N}^+$$

```
\neseqof{\Nat}
```

- Restricted types (those with invariants) can be typeset, with or without initialisation, using `\ritype` and `\rtype`, thus:

$$Partition = (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set}$$
$$\mathbf{inv}\ \textit{inv-Partition}(p)$$
$$Dict = \mathbb{B} \times (Letter \xrightarrow{m} Dict)$$
$$\mathbf{inv}\ \mathbf{true}$$
$$\mathbf{init}\ (\mathbf{true}, \{\,\})$$

```
\rtype{Partition}
  {\setof{(\setof{\Nat})}}
  {inv-Partition(p)}
\ritype{Dict}
  {\Bool \x (\mapof{Letter}{Dict})}
  {\true}
  {(\true,\emptymap)}
```

  Accompanying these commands are `\betweenTypeAndInvSkip` (default .5ex + .3ex − .2ex) and `\betweenInvAndInitSkip` (same default).

- Record types may also have invariants and initial states attached, using the `\inv` and `\init` commands within the `record` environment, thus:

$$D :: \begin{array}{l} day : Day \\ year : Year \end{array}$$
$$\mathbf{inv}\ (\textit{mk-}D(d,y)) \triangleq \textit{is-leapyr}(y) \vee d \leq 365$$
$$\mathbf{init}\ day = 40 \wedge year = 1962$$

```
\begin{record}{D}
  day : Day \\
  year : Year
\inv{(mk-D(d,y)) \DEF
  is-leapyr(y) \Or d \le 365}
\init{day=40 \And year=1962}
\end{record}
```

  To go with these are `\betweenRecordAndInvHook`, `\betweenInvAndInitHook`, `\betweenRecordAndInvSkip` (default .5ex + .2ex − .1ex), and `\betweenInvAndInitSkip` (same default).

11 Acknowledgements

Many people have passed on useful suggestions and comments about `vdm` and this documentation; many thanks to all of them. In particular I would like

to acknowledge the extensive testing done by Lynn Marshall while preparing her thesis, and her helpful comments and ideas, and the numerous worthwhile discussions with Cliff Jones. Cliff, in particular, deserves the highest commendation for bravery, in actually using these macros in his book. Thanks to David Carlisle for helping with the conversion to TEX 3.