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 authors of `vdm` to concentrate on the abstract 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. Letjs 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 sectionsnthen youjll know roughly what youjre 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:

[Sorry. Ignored `\beginvdm ... \endvdm`]

```
\beginvdm
\beginfndecptrs,om \\
\signature
 \setofOop \x \mapofOopObject \to \mapofOopObject

\If ptrs = \emptyset
\Then om
\Else   \Let gone = \setp \in ptrs | RC(om(p)) = 1 \In
   \Let om' = gone \dsub om \In
   \Let om'' = om' \owr
 \mapp \mapsto \chgom'(p)RCRC\minus 1
    | p \in ptrs \diff gone \In
   dec(\Union\set\elemsBODY(om(p)) | p \in gone, om'')
\Fi
\endfn
```

```
\beginop[DESTROYPTR]
\args Obj, Ptr : Oop
\ext \Wr OM : \mapofOopBasic_Object
\pre ptr \in  \elemsBODY(om(obj))
\post om = ~om \owr \map obj \mapsto
 \chgom(obj)BODYBODY \diff \setptr
\endop
\endvdm
```

2 Using vdmnGeneral 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,

```
  \beginvdm
....
  \endvdm
```

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. This is not the case if you are using PSLATEX, as that does distinguish text italic from math italic. 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*.

   you really want to use the inormalj inter-letter spacing, say .

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 TEXjs macro. An `@` is still an @ when not in math mode. Occasionally you may find that an @ in math mode *doesnjt* give you a subscript (particularly when used with moving arguments). Should this happen, you are advised to use TEXjs macro, e.g., `$x\sb0$`.

   If you donjt use underscores much, and you want to use _ for subscripts, you can say (and 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

[Sorry. Ignored `\beginvdm ... \endvdm`]
than subtractions, so on the whole this alteration should save effort. If you really want to do a single subtraction sign, use `*`. If you find the default is inappropriate, you can revert to the original behaviour using ; `*` is the inverse. Example: `a-b \ne\mathminus a-b` gives
[Sorry. Ignored `\beginvdm ... \endvdm`]
.

4. `|` gives you a
[Sorry. Ignored `\beginvdm ... \endvdm`]
and not a |. Do you see the difference? No? The former goes between things, e. g.,
[Sorry. Ignored `\beginvdm ... \endvdm`]
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 `|`.
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 isnjt. `~x` will give you a
[Sorry. Ignored `\beginvdm ... \endvdm`]
. For long identifiers, such as
[Sorry. Ignored `\beginvdm ... \endvdm`]
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 . By default it is defined to be (for the New Font Selection Scheme).

7. The following macros have been altered in a non-trivial way: *, * (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 , the indentation at the beginning of each paragraph. If you want to change this, change the value of , e.g.:

```
\setlength\VDMindent0cm
```

will make your specs come out flush left. This document has been typeset with equal to 3*. Similarly, the right hand margin is controlled by a parameter called . By default it is also set to .

You can have a particular line spacing in force within the `vdm` environment. The spacing within a `vdm` environment is dictated by the command. Note that this is *not* a length, but a command. By default it expands to 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\VDMbaselineskip0.8\baselineskip
```

for example.

3 Typesetting formulas

Most of the text you enter within `vdm` environments will be in TEXjs 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 , , , and for the operators

[Sorry. Ignored `\beginvdm ... \endvdm`]

. (To retain compatibility with a previous version, * , ,
and * are still provided, but 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 * and * have been re-defined to take arguments. For example, to get

[Sorry. Ignored `\beginvdm ... \endvdm`]

type

```
\existsx \in Sp(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 .
In addition, two new quantifiers, and *, have been added:
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] x *Sp(x) *x *Sp(x)
Additionally, to complement , there is . This is the so-called kiota-functionl that returns the unique value, if there is one:
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] x *Sp(x)

If you want to use the old versions of * and * they are available under the pseudonyms of and .

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,

[Sorry. Ignored `\beginvdm ... \endvdm`]

can be obtained with

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

If you need kStracheyl brackets, e.g., *Me*, place the material to appear within the brackets within `\term ...` , thus: `$M\terme$`.
A special control sequence, , is available for constants. To get, for example, *Yes*|*No*, type `\constYes|\constNo`.

If you donjt like the font that constants are set in, you can change them by redefining the command . By default it expands to .

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 donjt. If you need an odd equation to stand on its own, use the `formula` environment:

```
\beginformula
x = 10
\Or  \foralli \in \Nati \ne 10 \Implies i \ne x
\endformula
```

The `formula` environment is similar to displayed math mode, except: formulas are indented by , not , 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 kthensl and kelsesl with kifsl. In the following definitions, whenever you see the term *math-mode-expression*, you should type an expression as if in math mode, but you neednjt 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 *math-mode-expression* *math-mode-expression* *math-mode-expression*.
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] x*S S x *
rightside
If you nest s then you must enclose inner s within braces:
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] ... ...
You are advised to place the extra braces exactly as above; donjt let extraneous spaces intervene between the keywords and the braces.
The macro always starts a new line for the then and else parts. If you want TEX to try to choose line breaks, use instead:
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] a=b c=d+e p=q+r+s+t+u
let?in
constructions are done in a similar way: *math-mode-expression* *math-mode-expression*, and *math-mode-expression* *math-mode-expression*.
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] x=f(y,z) g(x)+h(x)
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] x=f(y,z) x\s\up5(2)
Notice that takes a second argument, which is part of the same iparagraphj, where 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 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 . If you were to try typesetting something like

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

youjd 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 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 in action:

[Sorry. Ignored `\beginformula ... \endformula`]

```
\Cases select(x)
\nil  & \emptyset \\
mk-Lst(hd,tl)  & \sethd \union \elemstl
\Otherwise x
\Endcases
```

Note the
 is a *separator* and not a *terminator*nyou donjt need one after the last item. Also, the can appear anywhere between the `\Cases` and the , 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 ; if you want them on the right, say . The scope of the and commands is the current group. By default, you get .

3.2.1 The `formbox` Environment

Occasionally you might find that you want to put a line break in a place that canjt handle
. For example, if you have a command and the rhs of a particular case is too big, you canjt 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) &
\beginformbox
long_predicate_with(foo) \\
  \And long_predicate_with(bar)
```

```
\endformbox
 ...
```

[Sorry. Ignored `\beginvdm ... \endvdm`]
Note the extras braces around the `formbox`; these are required to khidel the
from the .

3.3 Other General Points about Formulas

will Hr iwillj read ishouldj. *always* start a new line. Sometimes this is done in addition to some other function (as in the macro, where it delimits a particular case), but you should be able to use

almost anywhere to force a line break. Indeed, sooner or later youjll 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 youjll 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 command takes a single argument that controls how much the next line will be indented:
[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]
[t] a b 2 b a 1 d e

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

[Sorry. Ignored `\beginvdm ... \endvdm`]
[t] (a b b a) d 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 youjll have to work around it.

The , , etc., constructions are all unusual in that itjs impossible to typeset something sensibly to the right of them. For example, if you try

```
   \existsx \in S
 \If x=0 \Then S=Q \Else S=P \Fi
   \Or S=\emptyset
```

then youjll get

[Sorry. Ignored `\beginvdm ... \endvdm`]
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 youjll have to insert an or a depending on whether your comment might span one or more lines:

```
 [t]
        [Sorry. Ignored \beginvdm ... \endvdm]
[t] the condition is true do the true part "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 youjre in big trouble if you want to typeset a three-page . The only statement I can make to mitigate this is: you shouldnjt have expressions that complicated in the first placenwho do you expect to read them? Remember: ktruth is beautyl, so if your formulas are not beautiful, then chances are theyjre not true either.
```

4 Typesetting data types

The following table lists the primitive types and values available:

0,1,? 1,2,? ?,*1,0,1,? Rationals Real numbers , Truth Falsehood Nil		, , ,

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

```
\makeNewKeyword\maybemaybe
```

and henceforth is a valid control sequence that produces the text maybe. The text of the second argument to can be anything; it doesnjt have to match your control sequence name.

If you donjt like the font that keywords are set in, you can change it by redefining the command . By default it expands to .

The following type-related commands are provided:

Output	Input	
x	`\setofx`	set type constructor
a,b,c	`\seta,b,c`	set enumeration
*	*	the empty set
x	`\seqofx`	seq. type constructor

a,b,a,c	\seqa,b,a,c	seq. enumeration
		the empty sequence
xy	\mapofxy	map type constructor
xy	\mapintoxy	one-one map type
*p*x*	\mapp\mapsto x	map enumeration
		the empty map

Here are the relevant operators:

*	*				
*	*			*l*	\lenl
*	*			*l*	\hdl
*	*			*l*	\tll
	,;			*l*	\elemsl
	,;	*m*	\domm	*l*	\indsl
		m	\rngm	*l*	\Concl
		s	\Mins	*h,t*	\consh,t
	,;	*s*	\Maxs		
s	\cards				

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

```
\newMonadicOperator\invinv
```

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

You can define a new type using `\typetype-nametype`:

[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] Complex

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

[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t]

[Sorry. Ignored `\begincomposite ... \endcomposite`]

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

[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t]

[Sorry. Ignored `\begincomposite* ... \endcomposite*`]

[t]

[Sorry. Ignored `\beginvdm ... \endvdm`]

[t] Celsius

iRecordsj can be defined using the `record` environment:

```
\beginrecordrecord-type-name
field-name : field-type

?
\endrecord
```

The colons are used as sub-field separators.

[t]
[Sorry. Ignored `\beginvdm ... \endvdm`]
[t]
[Sorry. Ignored `\beginrecord ... \endrecord`]

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

```
\defrecordPERSON
NM : \seqofChar \\
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 ; if you want them on the right, say . The scope of the and commands are the current group. By default, you get .

Updating fields of composites using the *-function can be specified using :
[t]
[Sorry. Ignored `\beginvdm ... \endvdm`]
[t] pFEMman(q)

Notice that the *, parentheses, comma and * were inserted automatically.

5 How to Typeset Functions

Typesetting *-expressions is easy:
[t]
[Sorry. Ignored `\beginvdm ... \endvdm`]
[t] x,yx\s\up5(2)+y\s\up5(2)

As with *, * and , has a *-form that places the body of the function below and to the right:
[t]
[Sorry. Ignored `\beginvdm ... \endvdm`]
[t] x,y,z (x\s\up5(2)+y\s\up5(2)+z\s\up5(2))\s\up5(\F(1,2))

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

```
\beginfnname-of-function argument-list
\signaturesignature-of-function
*optional precondition*
*optional postcondition*
body of function (a *math-mode-expression*)
\endfn
```

See the third page for an example. The is optional and can be placed anywhere within the bodynit will always be typeset before the body. Useful macros within the are: and *, which yield and *. 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.

[Sorry. Ignored `\begin ... \end`]