# VDMJ Tool Support: User Guide

by
Nick Battle
Fujitsu UK

# 0. Document Control

## 0.1. Table of Contents

## 0.2. References

[1]    Wikipedia entry for The Vienna Development Method,
        http://en.wikipedia.org/wiki/Vienna_Development_Method
[2]    Wikipedia entry for Specification Languages, http://en.wikipedia.org/wiki/Specification_language

[3]     The VDM Portal, http://www.vdmportal.org/twiki/bin/view
[4]     The VDMTools VDM-SL Language Manual,
        http://www.vdmtools.jp/uploads/manuals/langmansl_a4E.pdf
[5]     The VDMTools VDM++ Language Manual,
        http://www.vdmtools.jp/uploads/manuals/langmanpp_a4E.pdf
[6]     Validated Designs for Object-oriented Systems, by John Fitzgerald et. al., http://www.vdmbook.com/
[7]     User Manual for the Overture Combinatorial Testing Plug-in, by Peter Gorm Larsen and Kenneth
        Lausdahl.
[8]     Overture, Open-source Tools for Formal Modelling, http://www.overturetool.org/
[9]     VDMJ Project, *SourceForge*, https://sourceforge.net/projects/vdmj/
[10]    *VDMJ Design Specification*, Nick Battle.
[11]    *Combinatorial Testing for VDM++.* Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle,
        SEFM-2010.
[12]    JUnit Test Framework, http://en.wikipedia.org/wiki/JUnit
[13]    The *rlwrap* manual page, http://linux.die.net/man/1/rlwrap
[14]    VDMJ Annotations Guide, 15/02/19.
[15]    Visual Studio Code extension for VDM language support , https://github.com/jonaskrask/vdm-vscode.
[16]    VDM Combinatorial Testing Guidelines, Nick Battle, Peter Gorm Larsen.

## 0.3.     Document History

| | | |
|---|---|---|
| Issue 0.1 | 10/10/08 | First release. |
| Issue 1.0 | 15/01/13 | Update for Overture 1.0.0 release and misc corrections |
| Issue 2.0 | 26/07/17 | Updated for VDMJ version 4 |
| Issue 2.1 | 14/03/19 | Updated for 4.3.0 annotations. |
| Issue 2.2 | 18/06/20 | Updated for script command. |
| Issue 2.3 | 06/01/21 | Removed DBGP section, updated threaded debugging. |
| Issue 2.4 | 03/03/21 | Updated vdmj.properties names. |
| Issue 2.5 | 29/09/21 | Added catch exception breakpoints |
| Issue 2.6 | 23/12/21 | Updated stdlib section for VDMJ 4.4.4 |
| Issue 2.7 | 23/02/22 | Remove references to -o option. |
| Issue 2.8 | 27/03/22 | Added external format reader and power set limits |
| Issue 2.9 | 10/06/22 | Update library loading details. |
| Issue 2.10 | 05/12/22 | Updated to add pause/interrupt keys, and RESULTs |
| Issue 2.11 | 10/01/23 | Updated vdmj.properties names. |
| Issue 2.12 | 07/02/23 | Updated for vdmj.plugins. |
| Issue 2.13 | 01/03/23 | Updated for VDMJ plugins, distribution and *vdmj.sh* |
| Issue 2.14 | 03/09/23 | Updated for maximal type |

## 0.4.     Copyright

# 1. Introduction

VDMJ [9] provides computer based tool support for the VDM-SL, VDM++ and VDM-RT specification languages, written in Java. The tool includes a parser, a type checker, an interpreter, a debugger, a proof obligation generator and a JUnit test framework. It is a command line tool only, though it is integrated with the Overture Eclipse GUI [8], and Visual Studio Code [15].

## 1.1. VDM

To quote from [1] and [2]:

A specification language is a formal language used in computer science. Unlike most programming languages, which are directly executable formal languages used to implement a system, specification languages are used during systems analysis, requirements analysis and systems design.

Specification languages are generally not directly executed. They describe the system at a much higher level than a programming language. Indeed, it is considered as an error if a requirement specification is cluttered with unnecessary implementation detail, because the specification is meant to describe the what, not the how.

The Vienna Development Method (VDM) is one of the longest-established Formal Methods for the development of computer-based systems. Originating in work done at IBM's Vienna Laboratory in the 1970s, it has grown to include a group of techniques and tools based on a formal specification language - the VDM Specification Language (VDM-SL). It has an extended form, VDM++, which supports the modelling of object-oriented and concurrent systems. Support for VDM includes commercial and academic tools for analyzing models, including support for testing and proving properties of models and generating program code from validated VDM models. There is a history of industrial usage of VDM and its tools and a growing body of research in the formalism has led to notable contributions to the engineering of critical systems, compilers, concurrent systems and in logic for computer science.

# 2.       Using VDMJ

## 2.1.       Installing VDMJ

The core of VDMJ is contained within one Java archive, with a name like *vdmj-<version>.jar*. This is used in the examples in the next section. But there is a suite of jars that can be combined with the main one, contributing new services, such as access to libraries and plugins. The entire suite is packaged into a *distribution* ZIP file, available from the main GitHub releases site[1].

To install the suite, download the distribution, *vdmj-suite-<version>-distribution.zip*, and unzip this into a local folder. The distribution contains all of the necessary jars, along with the documentation and a *bash* shell script called *vdmj.sh* to start VDMJ with most of the jars on the classpath. If you add the installation folder to your execution PATH, you should be able to start using the *vdmj.sh* script.

The remaining examples in this section show the raw execution of the core jar. In most cases, the same command line switches work the same way via the *vdmj.sh* script.

## 2.2.       Starting VDMJ

As mentioned above, the core of VDMJ is contained entirely within one jar file.  The jar file contains a MANIFEST that identifies the main class to start the tool, so the minimum command line invocation is as follows:

```
$ java -jar vdmj-4.n.n.jar
You did not specify any source files

$ java -jar vdmj-4.n.n.jar -help
Usage: VDMJ [-vdmsl | -vdmpp | -vdmrt] [<options>] [<files or dirs>]
-vdmsl: parse files as VDM-SL (default)
-vdmpp: parse files as VDM++
-vdmrt: parse files as VDM-RT
-v: show VDMJ jar version
-path: search path for files
-strict: use strict grammar rules
-r <release>: VDM language release
-t <charset>: select a console charset
-q: suppress information messages
-verbose: display detailed startup information
-c <charset>: select a file charset
-w: suppress warning messages
-i: run the interpreter if successfully type checked
-e <exp>: evaluate <exp> and stop
-c <charset>: select a file charset
-default <name>: set the default module/class
-pre: disable precondition checks
-post: disable postcondition checks
-inv: disable type/state invariant checks
-dtc: disable all dynamic type checking
-exceptions: raise pre/post/inv violations as <RuntimeError>
-measures: disable recursive measure checking
-annotations: enable annotation processing
-log <filename>: enable real-time event logging
-remote <class>: enable remote control
-simulation <class>: enable simulation control
-p: generate proof obligations
```

The tool should be invoked with either the *-vdmsl, -vdmpp* or *-vdmrt*  option to indicate the VDM

---

[1] https://github.com/nickbattle/vdmj/releases

dialect and parser required, though it defaults to *-vdmsl*. The *-r* option can be used to indicate the language release, currently *classic* or *vdm10*. The default is *vdm10*.

Normally, a specification will be loaded by identifying all of the VDM source files to include. If a directory is given, all the relevant source files within that directory will be loaded. The *-path* option may occur zero or more times, and defines alternative directories to search for files. At least one source file must be specified unless the *-i* option is used, in which case the interpreter can be started with no specification. Source files can be in a variety of document formats, as well as plain text (see section 2.3).

The parser allows some flexibility by default, but the *-strict* option will raise warnings for constructs in a specification which are strictly not legal. This is good for increasing compatibility with other tools.

If no *-i* option is given, the tool will parse and type check the specification files only, giving any errors and warnings on standard output, then stop. Warnings can be suppressed with the *-w* option. The *-q* option can be used to suppress the various information messages printed (this doesn't include errors and warnings).

The *-p* option will run the proof obligation generator and then stop, assuming the specification has no type checking errors.

For batch execution, the *-e* option can be used to identify a single expression to evaluate in the context of the loaded specification, assuming the specification has no type checking errors.

The *-c* and *-t* options allow the file and console character sets to be defined, respectively. This is to allow specifications written in languages other than the default for your system to be used (see section 6).

The *-default* option sets the default class or module name, otherwise the default is simply the first class or module encountered by the parser in the file list.

The *-pre, -post, -inv, -dtc* and *-measures* options can be used to disable precondition, postcondition, invariant, dynamic type checking and recursive measure checking, respectively. By default, all these checks are performed. The *-annotations* option enables annotation processing, which by default is disabled.

The *-log* option is for use with *-vdmrt*, and causes real-time events from the model to be written to the file name given. These are useful with the Overture Eclipse GUI, which has a plugin to display timing diagrams [8].

The *-remote* option is for enabling the remote control of a VDMJ process by another system, typically a test generator or a user interface. See section 3.2 of the VDMJ Design Specification [10] for details. Similarly, the *-simulation* option allows a class to be identified that works with VDM-RT to intercept the execution at various points.

The *-verbose* option gives more detailed timing information in version 4.

In addition to these command line options, various settings can be added to a vdmj.properties file (see section 5).

If *flat.def* contains a simple VDM-SL specification of the factorial function, called "f", the following illustrate ways to test the specification, with user input shown in **bold.** Note also that these same commands can be executed via the *vdmj.sh* script which comes with the VDMJ distribution (see 2.1). For example, the first command below would be **vdmj.sh -v**.

```
$ cat flat.def

functions
    f: int -> int
    f(a) == if a < 2 then 1 else a * f(a-1)
    pre a > 0


$  java -jar vdmj-4.n.n.jar -v
VDMJ version = 4.n.n build 230207
```

```
$  java -jar vdmj-4.n.n.jar -vdmsl flat.def
Parsed 1 module in 0.189 secs. No syntax errors
Warning 5012: Recursive function has no measure in 'DEFAULT' (flat.def) at
line 3:1
Type checked 1 module in 0.058 secs. No type errors and 1 warning


$ java -jar vdmj-4.n.n.jar -vdmsl -q -w flat.def
<quiet!>


$ java -jar vdmj-4.n.n.jar -vdmsl -verbose -w flat.def
Parsed 1 module in 0.076 secs. No syntax errors
Loaded ast-tc.mappings in 0.171 secs
Mapped 38 nodes with ast-tc.mappings in 0.002 secs
Type checked 1 module in 0.005 secs. No type errors and suppressed 1
warning


$ java -jar vdmj-4.n.n.jar -vdmsl -w -e "f(10)" flat.def
Parsed 1 module in 0.189 secs. No syntax errors
Type checked 1 module in 0.018 secs. No type errors and suppressed 1
warning
Initialized 1 module in 0.095 secs.
3628800
Bye


$ java -jar vdmj-4.n.n.jar -vdmsl -e "f(10)" -q -w flat.def
3628800


$ java -jar vdmj-4.n.n.jar -vdmsl -i -w flat.def
Parsed 1 module in 0.198 secs. No syntax errors
Type checked 1 module in 0.018 secs. No type errors and suppressed 1
warning
Initialized 1 module in 0.125 secs.
Interpreter started
> print f(10)
= 3628800
Executed in 0.023 secs.
> quit
Bye


$ java -jar vdmj-4.n.n.jar -vdmsl -p -w flat.def
Parsed 1 module in 0.182 secs. No syntax errors
Type checked 1 module in 0.023 secs. No type errors and suppressed 1
warning
Generated 2 proof obligations:

Proof Obligation 1: (Unproved)
f: function apply obligation in 'DEFAULT' (flat.def) at line 4:38
(forall a:int & pre_f(a) =>
  (not (a < 2) =>
    pre_f((a - 1))))

Proof Obligation 2: (Unproved)
f: recursive function obligation in 'DEFAULT' (flat.def) at line 3:1
(forall a:int & pre_f(a) =>
  (not (a < 2) =>
    measure_f(a) > measure_f((a - 1))))

Analysed 1 module in 0.107 secs. No POG errors
```

```
$ java -jar vdmj-4.n.n.jar -vdmsl -e "f(0)" -w flat.def
Parsed 1 module in 0.053 secs. No syntax errors
Type checked 1 module in 0.142 secs. No type errors and suppressed 1
warning
Initialized 1 module in 0.141 secs. No init errors
Execution: Error 4055: Precondition failure: pre_f in 'DEFAULT' (flat.def)
at line 5:11
        a = 0
In root context of pre_f(a) in 'DEFAULT' (console) at line 1:1
In root context of f(a) in 'DEFAULT' (console) at line 1:1
In root context of module scope
Bye


$ java -jar vdmj-4.n.n.jar -vdmsl -pre -e "f(0)" -w flat.def
Parsed 1 module in 0.22 secs. No syntax errors
Type checked 1 module in 0.024 secs. No type errors and suppressed 1
warning
Initialized 1 module in 0.097 secs.
1
Bye
```

## 2.3.    Command Line Recall

VDMJ does not automatically provide command line recall – the ability to quickly retrieve and perhaps change a previously entered command. But this feature can conveniently be added, either in Linux or Cygwin on Windows, by using the *rlwrap* command to start java [13]. For example:

```
$ rlwrap java -jar vdmj-4.n.n.jar -vdmsl -i MATH.java
Parsed 1 module in 0.189 secs. No syntax errors
Type checked 1 module in 0.013 secs. No type errors
Initialized 1 module in 0.031 secs.
Interpreter started
> p exp(1)
= 2.718281828459045
Executed in 0.018 secs.
> p exp(2)
= 7.38905609893065
Executed in 0.003 secs.
> p exp(1)                <-- Two up-arrows recalled this line
= 2.718281828459045
Executed in 0.002 secs.
> p exp(2)                <-- CTRL-R followed by (2 found this line
= 7.38905609893065
Executed in 0.003 secs.
>
```

The commands themselves are written to a file called *.java_history* in your home directory, which means that commands can be retrieved across VDMJ sessions. The location of the history file can be changed by setting the $RLWRAP_HOME environment variable. See [13] for details.

Note that the VDMJ distribution's *vdmj.sh* script will use *rlwrap* if it is installed on your machine.

## 2.4.    Source File Formats

VDM source files can be presented to VDMJ in a variety of formats and encodings. The embedding of a specification in another document (typically a design or requirement document) is useful as it allows

a single controlled source to be maintained[2]. The following formats are supported:

- **Plain text**. This is default format. Source files can be read using a variety of character encodings, as indicated by the *-c* command line option.

- **LaTeX**. Source files may be written in LaTeX so that they can subsequently be typeset to produce high quality printed documentation or PDF files. VDMJ will ignore any LaTeX markup, parsing only the text between markers `\begin{vdm_al}` and `\end{vdm_al}`. There can be several such marked sections in one document.

  LaTeX can be embedded in normal *.vdmsl files etc., or files called *.tex or *.latex are assumed to contain LaTeX.

  To preserve line number information, the lines outside of the VDM blocks are considered as blank lines in the specification. This means that errors reported on a given line will accurately reflect the line number in the LaTeX source document, but certain commands in VDMJ (like *coverage*) will show the blank lines in their output.

  Note that the start of a LaTeX source file has to be recognizable as LaTeX so that the parser can distinguish it from plain text. To achieve this, the first line in the file must be a LaTeX `\document`, `\section`, `\subsection` or a %comment. It is possible to parse LaTeX files containing different character encodings by using the *-c* command line option.

- **Word .doc**. Source files may be written in MS Word. The actual VDM source must be marked with `%%VDM%%` around the source lines, with each marker on a line by itself. All the sections marked this way will be extracted from the document by the parser. However, unlike the LaTeX parser, the lines outside of these blocks are ignored and do not play a part in the line numbering. This means that it can be difficult to tie an error message location to the line in the Word document, though the VDMJ *save* command will write the extracted source into a plain file, which helps. The support of international character sets in .doc files is limited, though files in plain ASCII will work. The preferred MS format for international character sets is .docx. The VDM source can appear in any font, and can have styles applied to the text (bold etc), all of which will be ignored by the parser.

- **Word .docx**. More recent versions of MS Word will save files in OOXML files with a .docx extension, and these are accepted by the VDMJ parser. The same marker technique and limitations apply as for .doc files, except that the support of international character sets is sane in .docx – the character encoding used for a .docx file is embedded in the file itself, and so there is no need to specify this via the *-c* option. The *save* command will write out the plain extracted source.

- **Open Document Format**. Various modern word processors can write files in ODF format (for example, *OpenOffice* and *Koffice*). These files have the extension .odt, and they are accepted by the VDMJ parser. The same marker techniques apply as for .docx files. The *save* command will write out the plain extracted source. Support for international character sets is provided, again without the need for the *-c* option.

- **AsciiDoc Format**. This is a text based format. See https://asciidoc.org. VDM sections are marked with an attribute, `{vdm}`.

- **Markdown Format**. This is also a text based format. See https://www.markdownguide.org. VDM sections are marked with an XML comment, `<!-- vdm -->`.

- **External Format**. VDMJ can be extended to read arbitrary document formats. To do this, the user must include an implementation of the *ExternalFormatReader* interface and set the vdmj.parser.external_readers property. Example PDF and CSV readers are available to copy.

Apart from plain text files (which is the default), VDMJ will recognise the various supported file formats above from the source file name extension: .tex/.latex, .doc, .docx, .odt., .adoc or .md/.markdown.

---

[2] This also supports *Literate Programming* – see http://en.wikipedia.org/wiki/Literate_programming

The extensions are not case sensitive.

It is occasionally useful to *conditionally* include source, depending on the dialect of VDM that is being parsed, or other environmental flags (Java properties). To enable this, the VDMJ parser includes an #ifdef convention, and will conditionally include or exclude lines when parsing plain text or LaTeX files. The symbols VDM_SL, VDM_PP and VDM_RT are pre-defined, depending on the dialect of the parser, and new symbols can be introduced with #define. All symbols are held as Java system properties (eg. set with the -D Java executable flag). For example:

```
#ifdef VDM_SL
values
        sl:nat = 123;
#else
instance variables
#ifdef VDM_PP
        pp:int := 123;
#else
        rt:int := 321;
#endif
#endif
```

As with LaTeX files, lines that are excluded with #ifdef or #ifndef are present as blank lines in the parsed specification, which enables line numbering to be accurate.

## 2.5.       Parsing, Type Checking, and Proof Obligations

All specification files loaded by VDMJ are parsed and type checked automatically. You can affect this with the -w, -strict and -r options; the type checker always uses "possible" semantics. If a specification does not parse and type check cleanly, the interpreter cannot be started and proof obligations cannot be generated (though warnings are allowed).

All warnings and error messages are printed on standard output, even with the *-q* option.

The Java program will return with an exit code of 0 if the specification is clean (ignoring warnings). Parser or type checking errors result in an exit code of 1.  The interpreter and PO generator always exit with a code of 0.

## 2.6.       The Interpreter

Assuming a specification does not contain any parse or type checking errors, the interpreter can be started by using the *-i* command line option.

The interpreter is an interactive command line tool that allows expressions to be evaluated in the context of the specification loaded.  The interpreter prompt is ">".  The following illustrates some of the interactive interpreter commands (explanation below. The *shmem* source code is in Appendix A):

```
$ java -jar vdmj-4.n.n.jar -vdmsl -i shmem.vdm
Parsed 1 module in 0.337 secs. No syntax errors
Warning 5000: Definition 'i' not used in 'M' (shmem.vdm) at line 129:7
Type checked 1 module in 0.109 secs. No type errors and 1 warning
Initialized 1 module in 0.091 secs.
Interpreter started

> help
init - re-initialize the specification
set [<pre|post|inv|dtc|measures|annotations> <on|off>] - set runtime checks
default <module> - set the default module name
modules - list the specification modules
files - list the specification files
plugins - list the loaded plugins
```

```
env - list the global symbols in the default environment
state - show the default module state
[p]rint <exp> - evaluate an expression
script <file> - run commands from file
assert <file> - check assertions from a file
break [<file>:]<line#> [<condition>] - create a breakpoint
break <function/operation> [<condition>] - create a breakpoint
trace [<file>:]<line#> [<exp>] - create a tracepoint
trace <function/operation> [<exp>] - create a tracepoint
catch [<exp list>] - create an exception catchpoint
remove <breakpoint#> - remove a trace/breakpoint
list - list breakpoints
coverage clear|write <dir>|merge <dir>|<filenames> - handle line coverage
latex|latexdoc [<files>] - generate LaTeX line coverage files
word [<files>] - generate Word HTML line coverage files
save [<files>] - generate external source extract files
runtrace <name> [start test [end test]] - run CT trace
debugtrace <name> [start test [end test]] - debug CT trace
savetrace [<file> | off] - save CT trace output
seedtrace <number> - seed CT trace random generator
runalltraces [<name>] - run all CT traces in class/module name
filter %age | <reduction type> - reduce CT trace(s)
pog [<function/operation>] - generate proof obligations
reload - reload specification files
help - list all commands available
[q]uit - close the session
```

**> -- Lines starting with "--" are ignored**
**> -- and lines that end with "\" are concatenated before execution**
```
print "abc" ^ \
      "def" ^ \
      "ghi"
= "abcdefghi"
Executed in 0.004 secs.
```

**> modules**
```
M (default)
```

**> state**
```
Q4 = [mk_M(<FREE>, 0, 9999)]
rseed = 87654321
Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)], [mk_M(<FREE>, 0,
9999)])
Q3 = [mk_M(<FREE>, 0, 9999)]
```

**> print rand(100)**
```
= 71
Executed in 0.0 secs.
```

**> print rand(100)**
```
= 44
Executed in 0.0 secs.
```

**> state**
```
Q4 = [mk_M(<FREE>, 0, 9999)]
rseed = 566044643
Memory = mk_Memory(566044643, [mk_M(<FREE>, 0, 9999)], [mk_M(<FREE>, 0,
9999)])
Q3 = [mk_M(<FREE>, 0, 9999)]
```

**> init**
```
Cleared all coverage information
Global context initialized
```

**> state**

```
Q4 = [mk_M(<FREE>, 0, 9999)]
rseed = 87654321
Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)], [mk_M(<FREE>, 0,
9999)])
Q3 = [mk_M(<FREE>, 0, 9999)]

> print rand(100)
= 71
Executed in 0.0 secs.

> print rand(100)
= 44
Executed in 0.0 secs.

> env
QuadrantLen2 = (nat1 * nat1 * Quadrant -> nat)
BestFit = (nat1 ==> bool)
inc = (() ==> ())
QuadrantLen0 = (Quadrant -> nat)
MQuadrantLen = (M * Quadrant -> nat)
add = (nat1 * nat1 * Quadrant -> Quadrant)
rand = (nat1 ==> nat1)
sizeof = (M -> nat1)
MAXMEM = 10000
delete = (M * Quadrant -> Quadrant)
inv_M = (M +> bool)
CHUNK = 100
FirstFit = (nat1 ==> bool)
least = (nat1 * nat1 -> nat1)
TryFirst = (nat ==> nat)
pre_DeleteOne = (Memory +> bool)
pre_add = (nat1 * nat1 * Quadrant +> bool)
spacefor = (nat1 * Quadrant -> nat1)
inv_Memory = (Memory +> bool)
fragments = (Quadrant -> nat)
combine = (Quadrant -> Quadrant)
seed = (nat1 ==> ())
DeleteOne = (() ==> ())
bestfit = (nat1 * Quadrant -> nat1)
Reset = (() ==> ())
QuadrantLen = (nat1 * Quadrant -> nat)
main = (nat1 * nat1 ==> seq of ((<BEST> | <FIRST> | <SAME>)))
TryBest = (nat ==> nat)
init_Memory = (Memory +> bool)

> pog
Generated 64 proof obligations:

Proof Obligation 1: (Unproved)
M: total function obligation in 'M' (shmem.vdm) at line 17:25
(forall mk_M($any1, a, b):M &
  is_(inv_M(mk_M($any1, a, b)), bool))

Proof Obligation 2: (Unproved)
M: invariant satisfiability obligation in 'M' (shmem.vdm) at line 13:1
exists mk_M(-, a, b) : M & (b >= a)

Proof Obligation 3: (Unproved)
Memory: total function obligation in 'M' (shmem.vdm) at line 25:40
(forall mk_Memory($any1, q3, q4):Memory &
  is_(inv_Memory(mk_Memory($any1, q3, q4)), bool))
...
>
```

This example shows a VDM-SL specification called *shmem.vdm* being loaded. The *help* command

lists the interpreter commands available. Note that several of them regard the setting of breakpoints, which is covered in the next section.

The *modules* command lists the names of the modules loaded from the specification. In this example there is only one, called "M". One of the modules is identified as the default; names in the default module do not need to be qualified (so you can say *print xyz* rather than *print M`xyz*). The default module can be changed with the *default* command (or the *-default* command line option).

The *state* command lists the content of the default module's state. This can be changed by operations, as can be seen by the two calls to *rand* which change the *rseed* value in the state (a pseudo-random number generator). The *init* command will re-initialize the state to its original value, illustrated by the fact that two subsequent calls to *rand* return the same results as the first two did.

The *print* command can be used to evaluate any expression.

The *env* command lists all the values in the global environment of the default module. This shows the functions, operations and constant values defined in the module. Note that it includes invariant, initialization and pre/postcondition functions.

The *pog* command (proof obligation generator) generates a list of proof obligations for the specification.

The *assert* command (illustrated below) can take a list of assertions from a file, and execute each of them in turn, raising an error for any assertion which is false. The assertions in the file must be simple boolean expressions, one per line:

```
$ cat rand.assertions
rand(100) = 71
rand(100) = 44
$

$ java -jar vdmj-4.n.n.jar -vdmsl -i shmem.vdm
Parsed 1 module in 0.339 secs. No syntax errors
Warning 5000: Definition 'i' not used in 'M' (shmem.vdm) at line 129:7
Type checked 1 module in 0.059 secs. No type errors and 1 warning
Initialized 1 module in 0.08 secs.
Interpreter started
>
> assert rand.assertions
PASSED all 2 assertions from rand.assertions
> assert rand.assertions
FAILED: rand(100) = 71
FAILED: rand(100) = 44
FAILED 2 and passed 0 assertions from rand.assertions
> init
Cleared all coverage information
Global context initialized
> assert rand.assertions
PASSED all 2 assertions from rand.assertions
> quit
Bye
```

The example is an assertion file which states that the first and second invocations of the *rand* operation will return 71 and 44 respectively. This is seen to be true the first time the assertions are run, but (having changed the state) the second time they are executed, both assertions fail (the actual pseudo-random values returned are different). Using the *init* command resets the state, so the assertions work correctly again.

The *script* command has the same form as the *assert* command, but instead of checking assertions, the file is considered as a sequence of commands (in exactly the same form as you could enter at the console) that are executed automatically:

```
> script startup
startup> init
```

```
Global context initialized
Executed in 0.004 secs.
startup> print Reset()
= ()
Executed in 0.014 secs.
startup> break 42
Created break [1] in 'M' (shmem.vdm) at line 42:18
42:     sizeof(m) == m.stop - m.start + 1;
END startup
```

Most commands can be added to a script, and you can call a script within a script. When the script completes, control is returned to the console; the name of the script file replaces the prompt as script commands are executed. Breakpoints can be created or removed in a script, but if an execution is performed (via a *print* command), the script will interact with the console to handle commands at the breakpoint. When you issue a *continue* at the console, the script will continue.

The *files* command will list the file names that comprise the current loaded specification. The *plugins* command lists the loaded plugins, see 2.11. The *reload* command will re-parse and check the specification files currently loaded. Note that if there are any errors in the parse or type check of the files, the interpreter will exit after *reload.*

If the files currently loaded are changed outside VDMJ, an indication will be printed between command prompts, until the *reload* or *load* commands are used to refresh the files:

```
>
File test.vpp has changed
>
File test.vpp has changed
> reload
Parsed 1 class in 0.0 secs. No syntax errors
Type checked 1 class in 0.0 secs. No type errors
Initialized 1 class in 0.0 secs.
Interpreter started
>
>
```

The *save* command will extract VDM source from MS Word or ODF source files and write it to a plain text file. This is useful when debugging sources embedded in these formats, since line number information does not relate to the source's position in the original document.

```
> files
sortingWithCombineSubfamily.odt
> save
Extracted source written to sortingWithCombineSubfamily.odt.vdmsl
>
```

The *set* command allows you to change the runtime checks in force (this is the same as the *-pre, -post, -inv, -dtc, -measures* and *-annotations* command line options). Using the command without arguments will list the current settings.

```
> set
Preconditions are enabled
Postconditions are enabled
Invariants are enabled
Dynamic type checks are enabled
Measure checks are enabled
Annotations are enabled
>
```

Most of the interpreter commands for a VDM++ specification are the same as for VDM-SL, with the following differences:

```
$ java -jar vdmj-4.n.n.jar -vdmpp -i
Initialized 1 class in 0.183 secs.
Interpreter started
> help
init - re-initialize the specification
set [<pre|post|inv|dtc|measures|annotations> <on|off>] - set runtime checks
default <class> - set the default class name
classes - list the specification classes
files - list the specification files
plugins - list the loaded plugins
env - list the global symbols in the default environment
[p]rint <exp> - evaluate an expression
script <file> - run commands from file
assert <file> - check assertions from a file
threads - list the running threads
create <id> := <exp> - create a named variable
break [<file>:]<line#> [<condition>] - create a breakpoint
break <function/operation> [<condition>] - create a breakpoint
trace [<file>:]<line#> [<exp>] - create a tracepoint
trace <function/operation> [<exp>] - create a tracepoint
catch [<exp list>] - create an exception catchpoint
remove <breakpoint#> - remove a trace/breakpoint
list - list breakpoints
coverage clear|write <dir>|merge <dir>|<filenames> - handle line coverage
latex|latexdoc [<files>] - generate LaTeX line coverage files
word [<files>] - generate Word HTML line coverage files
save [<files>] - generate external source extract files
runtrace <name> [start test [end test]] - run CT trace
debugtrace <name> [start test [end test]] - debug CT trace
savetrace [<file> | off] - save CT trace output
seedtrace <number> - seed CT trace random generator
runalltraces [<name>] - run all CT traces in class/module name
filter %age | <reduction type> - reduce CT trace(s)
pog [<function/operation>] - generate proof obligations
reload - reload specification files
help - list all commands available
[q]uit - close the session
>
```

The *classes* command replaces the *modules* command, but is similar in that it lists the names of the classes loaded, identifying the default.  The *threads* command lists the running threads. The *default* command is used to change the default class.  There is no *state* command because VDM++ objects contain state within instances of themselves as instance variables.  The state values for any object can be printed with the *print* command.  The *init* and *env* commands do the same thing, re-initializing static class members and printing the names and types of public static values accessing from the global context. The *create* command can be used to create a new (synthetic) public static of a given name, which can then be used in further evaluations.

The following example illustrates:

```
$ cat test.vpp
class A

instance variables
  si:int;
  sj:int;
  public static sk:int := 123;

operations

public op: int * int ==> int
  op(i, j) ==
  ( si := i;
```

```
        sj := j;
        sk := i + j;
        return sk;
    );

    public static test: () ==> int
        test() == let a = new A() in a.op(1, 2);

    end A
```

```
$ java -jar vdmj-4.n.n.jar -vdmpp -i test.vpp
Parsed 1 class in 0.236 secs. No syntax errors
Warning 5001: Instance variable 'si' is not initialized in 'A' (test.vpp)
at line 4:3
Warning 5001: Instance variable 'sj' is not initialized in 'A' (test.vpp)
at line 5:3
Type checked 1 class in 0.02 secs. No type errors and 2 warnings
Initialized 1 class in 0.052 secs.
Interpreter started
> classes
A (default)
> state
Command is only availble for VDM-SL
> p new A()
= A{#1, si:=undefined, sj:=undefined, sk:=123}
Executed in 0.026 secs.
> env
test() = (() ==> int)
sk = 123
> p test()
= 3
Executed in 0.0070 secs.
> env
test() = (() ==> int)
sk = 3
> p new A().op(123,456)
= 579
Executed in 0.0090 secs.
> env
test() = (() ==> int)
sk = 579
> create a := new A()
> env
test() = (() ==> int)
sk = 579
> p a.op(111,222)
= 333
Executed in 0.0020 secs.
> quit
Bye
```

Executing a specification written in the VDM-RT dialect is essentially the same as with VDM++. There are two additional commands, *log* and *validate*, which are used to indicate where real-time events from the interpreter should be written and subsequently validates them:

```
> help
classes - list the loaded class names
threads - list active threads
default <class> - set the default class name
create <id> := <exp> - create a named variable
log [<file> | off] - log RT events to file
validate <file> - validate RT log events
```

...

By default, event logging is turned off. But logging can be enabled to the console by using *log* with no arguments, or to a file using *log <filename>*. Logging can subsequently be turned off again by using *log off*. For example, the following can be used to see the real time events for the initialization of a VDM-RT specification:

```
> log
RT events now logged to the console

> init
Cleared all coverage information
ThreadCreate -> id: 1 period: false  objref: nil clnm: nil  cpunm: 0 time:
0
ThreadSwapIn -> id: 1 objref: nil clnm: nil  cpunm: 0 overhead: 0 time: 0
DeployObj -> objref: 1 clnm: "A" cpunm: 0 time: 0
OpRequest -> id: 1 opname: "A(nat)" objref: 1 clnm: "A" cpunm: 0 async:
false time: 0
OpActivate -> id: 1 opname: "A(nat)" objref: 1 clnm: "A" cpunm: 0 async:
false time: 0
OpCompleted -> id: 1 opname: "A(nat)" objref: 1 clnm: "A" cpunm: 0 async:
false time: 0
DeployObj -> objref: 2 clnm: "A" cpunm: 0 time: 0
OpRequest -> id: 1 opname: "A(nat)" objref: 2 clnm: "A" cpunm: 0 async:
false time: 0
OpActivate -> id: 1 opname: "A(nat)" objref: 2 clnm: "A" cpunm: 0 async:
false time: 0
OpCompleted -> id: 1 opname: "A(nat)" objref: 2 clnm: "A" cpunm: 0 async:
false time: 0
CPUdecl -> id: 1 expl: true sys: "SYS" name: "cpu1" time: 0
CPUdecl -> id: 2 expl: true sys: "SYS" name: "cpu2" time: 0
DeployObj -> objref: 6 clnm: "SYS" cpunm: 0 time: 0
DeployObj -> objref: 1 clnm: "A" cpunm: 1 time: 0
DeployObj -> objref: 2 clnm: "A" cpunm: 2 time: 0
BUSdecl -> id: 1 topo: {1,2} name: "bus" time: 0
ThreadSwapOut -> id: 1 objref: nil clnm: nil  cpunm: 0 overhead: 0 time: 0
ThreadKill -> id: 1 cpunm: 0 time: 0
Global context initialized
>

> log off
RT event logging disabled
> validate saved.log
Enable annotations first, using 'set annotations on'
```

Testing a specification by evaluating expressions is useful, but it is not certain that this will exercise all parts of the specification. To help determine the coverage of the tests executed against a specification, VDMJ keeps a record of which parts of of the specification have been executed. The overall code coverage can then be displayed with the *coverage* command, as the following example illustrates.

```
class A
functions
public f: int * int -> real
      f(x, y) ==
            x / y
      pre x < y;

operations
public g: int * int ==> real
      g(x, y) ==
            return x + y
```

```
end A


Parsed 1 class in 0.2 secs. No syntax errors
Type checked 1 class in 0.0080 secs. No type errors
Initialized 1 class in 0.0050 secs.
Interpreter started
> coverage
Test coverage for coverage.vpp:

  class A
  functions
  public f: int * int -> real
      f(x, y) ==
-           x / y
-     pre x < y;

  operations
  public g: int * int ==> real
      g(x, y) ==
-           return x + y

  end A

Coverage = 0%
>


> p new A().f(1,2)
= 0.5
Executed in 0.036 secs.
>
> coverage
Test coverage for test.vpp:

  class A
  functions
  public f: int * int -> real
      f(x, y) ==
+           x / y
+     pre x < y;

  operations
  public g: int * int ==> real
      g(x, y) ==
-           return x + y

  end A

Coverage = 66%
>
```

The *coverage* command displays the source code of the loaded specification (by default, all source files are listed), with "+" and "-" signs in the left hand column indicating lines which have been executed or not, respectively. In the example above, there are only three executable lines. Initially, none of them have been executed, and they all show a "-" sign in the left hand column. After the execution of the "f" function, its lines (including the precondition) are labelled "+", and the percentage coverage of the source file is displayed.

Typically, the testing of a specification will be incremental, and so it is convenient to be able to "save" the coverage achieved in each test session, and subsequently merge the results together. This can be achieved with the *write <dir>* and *merge <dir>* options to the *coverage* command. The *write* option saves the current coverage information in *<dir>* for each specification file loaded; the *merge* option

reads this information back, and merges it with the current coverage information. For example, each day's test coverage could be written to a separate "day" directory, and then all the days merged together for review of the overall coverage at the end.

Detailed coverage information is available with the *latex* and *latexdoc* commands. These write LaTeX files with parts of the specification highlighted where they have not been executed. The LaTeX output also contains a table of percentage cover by module/class and the number of times functions and operations were hit during the execution. The *latexdoc* command is the same, except that output files are wrapped in LaTeX document headers. The LaTeX output files are written to the same directory as source files, one per source file, with the extension .tex.

```
> latex
Latex coverage written to coverage.vpp.tex
>
```

Similarly, the *word* command will write detailed coverage information in MS Word HTML format. These files are standard HTML (and can be opened with a browser), but they use various styles which allow them to be read by MS Word, so you can open the files with the program as if they were normal .doc files.

```
> word
Word HTML coverage written to coverage.vpp.doc
>
```

Coverage information is reset when a specification is loaded or initialized, or when the command *coverage clear* is executed, otherwise coverage is cumulative. If several files are loaded, the coverage for just one source file can be listed with *coverage <file>*, *latex <file>* or *word <file>*.

## 2.6.1. Suppressing Runtime Checks

By default, the interpreter will evaluate all preconditions, postconditions, state and type invariants, perform dynamic type checking when values are manipulated in a specification, and check measures for recursive functions. These checks are valuable, and form an important part of the verification of a specification, which is why they are performed by default.

However, the checks also take a certain amount of processing power and result in a slower evaluation of expressions or tests. If you are certain that a specification will not violate the checks, you can use the *-pre, -post, -inv, -dtc, -measures* and *-annotations* options to selectively control them. These can either be passed on the command line to VDMJ, or issued at any time via the *set* command.

The *-pre* and *-post* options suppress the evaluation of both function and operation pre- and postconditions. The *-inv* option suppresses the evaluation of type invariants, state invariants (in VDM-SL) and class invariants (in VDM++ and VDM-RT). The *-dtc* option (*dynamic type checking*) is equivalent to *-inv* plus suppression of the checks that occur whenever a value of one type is assigned to another (eg. in parameter passing, result returns, in "let" definitions and binds, variable initializers etc). The *-measures* option suppresses recursive function measure checking. The *-annotations* option controls annotation processing (see below); if this option is changed, the specification must be re-parsed using the *reload* command afterwards.

## 2.7. Debugging

## 2.7.1. Single-threaded Debugging

As well as evaluating expressions and displaying the value of state variables, the interpreter is able to stop execution part way through an evaluation, and allow the evaluation to be single-stepped or traced, and intermediate values displayed.

Debugging is always enabled in VDMJ, so to perform debugging it is only necessary to set

breakpoints or tracepoints.  A breakpoint stops execution at a statement or expression, and enters a command line state where single-stepping can be performed; a tracepoint on a statement or expression does not stop execution, but prints out the value of an expression and carries on.

Using the *class A* example from section  2.6, we can stop on the *op* operation as follows:

```
> break op
Created break [1] in 'A' (test.vpp) at line 9:1
12:    ( si := i;
> p test()
Stopped break [1] in 'A' (test.vpp) at line 9:1
12:    ( si := i;
[MainThread]> help
step - step one expression/statement
next - step over functions or operations
out - run to the return of functions or operations
...
trace [<file>:]<line#> [<exp>] - create a tracepoint
trace <function/operation> [<exp>] - create a tracepoint
remove <breakpoint#> - remove a trace/breakpoint
list - list breakpoints
--
Stopped break [1] in 'A' (test.vpp) at line 9:1
9:  ( si := i;
MainThread> stack
Stopped at break [1] in 'A' (test.vpp) at line 9:1
        j = 2
        i = 1
        self = A{#1, si:=undefined, sj:=undefined, sk:=123}
In object context of op(i, j) in 'A' (test.vpp) at line 19:34
In context of let statement in 'A' (test.vpp) at line 19:15
In class context of test() in 'A' (console) at line 1:1
In root context of global static scope
--
Stopped break [1] in 'A' (test.vpp) at line 9:1
9:  ( si := i;
[MainThread]> source
7:
8:  operations
9:
10:  public op: int * int ==> int
11:    op(i, j) ==
12:>>  ( si := i;
13:      sj := j;
14:      sk := i + j;
15:      return sk;
16:    );
17:
[MainThread]> down
In context of let statement in 'A' (test.vpp) at line 19:15
--
Stopped break [1] in 'A' (test.vpp) at line 9:1
9:  ( si := i;
[MainThread]> stack
Stopped at break [1] in 'A' (test.vpp) at line 9:1
        a = A{#1, si:=undefined, sj:=undefined, sk:=123}
In context of let statement in 'A' (test.vpp) at line 19:15
In class context of test() in 'A' (console) at line 1:1
In root context of global static scope
--
Stopped break [1] in 'A' (test.vpp) at line 9:1
9:  ( si := i;
[MainThread]> print a
a = A{#1, si:=undefined, sj:=undefined, sk:=123}
```

```
--
Stopped break [1] in 'A' (test.vpp) at line 9:1
9:  ( si := i;
MainThread> step
10:  sj := j;
MainThread> step
11:  sk := i + j;
MainThread> step
12:  return sk;
MainThread> step
= 3
Executed in 303.669 secs.
>
> list
break [1] in 'A' (test.vpp) at line 9:1
12:    ( si := i;
> break op i < 10
Overwriting [1] in 'A' (test.vpp) at line 9:1
Created break [2] when "(A`i < 10)" in 'A' (test.vpp) at line 9:1
9:  ( si := i;
> print new A().op(1,2)
Stopped break [2] when "i < 10" in 'A' (test.vpp) at line 9:1
12:    ( si := i;
[MainThread]> continue
= 3
Executed in 21.217 secs.
> print new A().op(100,200)
= 300
Executed in 0.0020 secs.
>
```

Breakpoints can either be set on an operation name (in the default module or class, or using a fully qualified name), or at a specific line in the default file or a specific file using [<file>:]<line>. Optionally, after giving the location of a breakpoint, you can specify a boolean condition which will be evaluated whenever the breakpoint is reached, and only stops if it is true. It is also possible to conditionally stop at a breakpoint after a number of "hits" of that breakpoint. For example:

```
> break 9 = 4
Overwriting [2] in 'A' (test.vpp) at line 9:1
Created break [3] when "hits = 4" in 'A' (test.vpp) at line 9:1
9:  ( si := i;
> break 9 > 4
Overwriting [3] in 'A' (test.vpp) at line 9:1
Created break [4] when "hits > 4" in 'A' (test.vpp) at line 9:1
9:  ( si := i;
> break 9 >= 4
Overwriting [4] in 'A' (test.vpp) at line 9:1
Created break [5] when "hits >= 4" in 'A' (test.vpp) at line 9:1
9:  ( si := i;
> break 9 mod 4
Overwriting [5] in 'A' (test.vpp) at line 9:1
Created break [6] when "hits multiple of 4" in 'A' (test.vpp) at line 9:1
9:  ( si := i;
> break 9 i > j or i+j > 10
Overwriting [6] in 'A' (test.vpp) at line 9:1
Created break [7] when "((A`i > A`j) or ((A`i + A`j) > 10))" in 'A'
(test.vpp) at line 9:1
9:  ( si := i;
>
```

The first four examples would stop at line 12 of the specification when the hit count (the number of times the breakpoint is reached since the current execution started) is precisely four; when it is

greater than four; when it is four or greater; and when the hit count modulo 4 is zero (ie. 4, 8, 12, etc) respectively. The last breakpoint stops on line 12 when i is greater than j or the sum of i and j is greater than 10.

The other debugger commands are fairly standard, and do what you would expect. You can display the call *stack*, move *up* and *down* stack frames and evaluate expressions when a breakpoint is reached. You can examine the *source* for the current breakpoint. You can use *step* (or *next* or *out*, to step over and out of functions/operations). You can *list* and *remove* breakpoints by number.

Tracepoints are similar to breakpoints, except there is an implicit *continue* command after they have printed the value of the expression they contain. The trace expression is evaluated and printed *before* the expression or statement is executed, for example:

```
> trace 14 mk_(i, j, sk)
Created trace [1] show "mk_(i, j, sk)" in 'A' (test.vpp) at line 14:5
14:        sk := i + j;
> p test()
mk_(i, j, sk) = mk_(1, 2, 123) at [1]
= 3
Executed in 0.0040 secs.
>
```

Note that new breakpoints and tracepoints can be set when stopped in the debugger (ie. when the command prompt is "[thread-name-n]>"). Setting a trace or breakpoint at the same location as an existing one replaces the existing one silently.

Some commonly used commands can be abbreviated to a single letter: s for step, n for next, o for out, p for print, c for continue.

As well as at breakpoints, the debugger is also entered when a specification raises runtime faults. This simulates the existence of a breakpoint at the point of failure, and allows the stack environment to be examined to see what caused the problem. Of course any attempt to continue the specification execution, including single stepping commands, will not work as the specification has terminated – these cause the debugger to exit back to the normal command line state instead.

```
> p new A().f(-1,0)
Error 4134: Infinite or NaN trouble in 'A' (coverage.vpp) at line 5:11
Stopped in 'A' (coverage.vpp) at line 5:11
5:              x / y
[MainThread]> source
1:   class A
2:   functions
3:   public f: int * int -> real
4:       f(x, y) ==
5:>>              x / y
6:       pre x < y;
7:
8:   operations
9:   public g: int * int ==> real
10:       g(x, y) ==
11:              return x + y
[MainThread]> p x
x = -1
[MainThread]> p y
y = 0
[MainThread]> continue
Runtime: Error 4134: Infinite or NaN trouble in 'A' (coverage.vpp) at line
5:11
>
```

Catchpoints can be set to catch exceptions thrown by a VDM specification using the "exit" statement. They are similar to breakpoints, but they are not located at a particular line. Rather, in effect, they set a breakpoint on every "exit" statement in the specification.

---

By default, catchpoints will stop on every "exit" statement, but they can optionally take a list of values, and will only stop when those values are thrown by an exit statement. In the case of VDM++ and VDM-RT, catching the name of a class will stop if any instance of that class is thrown.

```
> catch
Created catch [1] (all exceptions)
> print op()
Stopped catch [1] (all exceptions)
11:        exit 123;
MainThread> s
5:              return x
MainThread> s
= 123
Executed in 5.447 secs.
>

> catch 456 789
Created catch [1] value = 456
Created catch [2] value = 789
> p op()
= 123
Executed in 0.006 secs.
>
```

Here, the second example did not trap because the catch command created two catchpoints that would stop when the values 456 and 789 were thrown, respectively. Since the exit statement throws 123, the evaluation completes.

The examples above are all inside a single module or class called "A", and by default all variable names are implicitly qualified with that name, so you don't have to type "A`x" to refer to a variable. However, when debugging more complex multi-class specifications, the debugger may break in a class that is different to the current default. Therefore, the debugger automatically sets the default class or module to the one the breakpoint is located in. At the end of a debugging session, the default class or module remains as the one set by the last breakpoint – it can be changed with the "default" command.

It is sometimes useful to know the return value of a function or an operation, especially if the value is not assigned to a variable by the caller. This information is provided as a "RESULT" value that is displayed when you step into and then out of a function or operation. The value also appears temporarily in the top stack frame, until you resume. For example:

```
functions
    f: nat * nat -> nat
    f(a, b) ==
        g(a) + g(b);   -- Breakpoint here

    g: nat -> nat
    g(a) ==
        a * 2;


> break 4
Created break [1] in 'DEFAULT' (test.vdm) at line 4:9
4:        g(a) + g(b);
> p f(1,2)
Stopped break [1] in 'DEFAULT' (test.vdm) at line 4:9
4:        g(a) + g(b);
MainThread> s
8:        a * 2;
MainThread> s
RESULT g(a) = 2
Stopped break [1] in 'DEFAULT' (test.vdm) at line 4:9
4:        g(a) + g(b);
MainThread> s
8:        a * 2;
```

```
MainThread> out
RESULT g(b) = 4
4:              g(a) + g(b);
MainThread> s
= 6
Executed in 12.018 secs.
>
```

Notice that after the first invocation of "g(a)", stepping past the end says "RESULT g(a) = 2" and then again after the "g(b)" call, using "out" reports that "RESULT g(b) = 4".

## 2.7.2.    VDM++ Multi-threaded Debugging

The examples in section 2.6.1 cover the debugging of a simple specification, either one written in VDM-SL, or one in VDM++ with no object threads. Therefore there is a single thread of control and the breakpoint prompt is always the same (usually [MainThread-7] – the number is decided by Java).

VDM++ specifications can create multiple threads, and the debugger provides some support for multi-threaded debugging. VDM-RT specifications can give even more information about threading (see 2.4.3 for a discussion of the more powerful VDMJC debugging client).

The basic principles and most debugging commands for multi-threaded debugging are the same as the single threaded case. There are two extra commands available, "thread" and "threads", which allow a thread to be selected and display the status of all active threads, respectively.

When a breakpoint is reached, all threads will be suspended. Single stepping will then advance the currently scheduled thread, while the others remain suspended until the scheduler selects them. Using the "continue" command will resume all threads. Single stepping may flip to one of the other threads, which can be confusing. The current thread name is always shown in the debugger prompt.

If any thread encounters a runtime error, then all threads are aborted. If the main thread terminates, all threads are aborted.

The following debug session illustrates some of the issues – the specification contains three threads, plus the main thread, all of which print out their thread IDs regularly (see the listing at the end of the section):

```
> break 9
Created break [1] in 'A' (test.vpp) at line 9:13
9:              for x = 1 to 10000 do skip
> p new A().run()
Thread 1...
Stopped break [1] in 'A' (test.vpp) at line 9:13
9:              for x = 1 to 10000 do skip
MainThread> threads
11: MainThread   break [1] in 'A' (test.vpp) at line 9:13
12: Object-A-12  (not started)
13: Object-A-13  (not started)
14: Object-A-14  (not started)

Stopped break [1] in 'A' (test.vpp) at line 9:13
9:              for x = 1 to 10000 do skip
MainThread> c
Thread 12...
Stopped break [1] in 'A' (test.vpp) at line 9:13
9:              for x = 1 to 10000 do skip
Object-A-12> c
Thread 13...
Stopped break [1] in 'A' (test.vpp) at line 9:13
9:              for x = 1 to 10000 do skip
Object-A-13> c
Thread 14...
Stopped break [1] in 'A' (test.vpp) at line 9:13
```

```
9:                   for x = 1 to 10000 do skip
Object-A-14> c
Thread 1...
Stopped break [1] in 'A' (test.vpp) at line 9:13
9:                   for x = 1 to 10000 do skip
MainThread> threads
11: MainThread   break [1] in 'A' (test.vpp) at line 9:13
12: Object-A-12  in 'A' (test.vpp) at line 9:35
13: Object-A-13  in 'A' (test.vpp) at line 9:35
14: Object-A-14  in 'A' (test.vpp) at line 9:35

Stopped break [1] in 'A' (test.vpp) at line 9:13
9:                   for x = 1 to 10000 do skip
MainThread> stack
Stopped [MainThread] in 'A' (test.vpp) at line 9:13
In context of block statement in 'A' (test.vpp) at line 7:9
In context of block statement in 'A' (test.vpp) at line 5:5
     A`self = A{#1}
In object context of op() in 'A' (test.vpp) at line 17:9
In context of block statement in 'A' (test.vpp) at line 15:5
In object context of run() in 'A' (console) at line 1:1
In root context of global static scope

Stopped break [1] in 'A' (test.vpp) at line 9:13
9:                   for x = 1 to 10000 do skip
MainThread> thread 14
9:                   for x = 1 to 10000 do skip
Object-A-14> stack
Stopped [Object-A-14] in 'A' (test.vpp) at line 9:35
     x = 9989
In context of for index in 'A' (test.vpp) at line 9:13
In context of block statement in 'A' (test.vpp) at line 7:9
In context of block statement in 'A' (test.vpp) at line 5:5
     A`self = A{#4}
In object context of op() in 'A' (test.vpp) at line 21:5
In object context of thread() in 'A' (test.vpp) at line 16:9
In object context of Thread 14, self #4, class A in 'A' (test.vpp) at line
16:9
In root context of global static scope

Object-A-14> c
Thread 12...
Stopped break [1] in 'A' (test.vpp) at line 9:13
9:                   for x = 1 to 10000 do skip
Object-A-12> c
Thread 13...
Stopped break [1] in 'A' (test.vpp) at line 9:13
9:                   for x = 1 to 10000 do skip
Object-A-13> c
Thread 14...
Stopped break [1] in 'A' (test.vpp) at line 9:13
9:                   for x = 1 to 10000 do skip
Object-A-14> stop
= undefined
Executed in 573.929 secs.
>
```

The operation, "op" loops, calling the IO class to print out its thread ID, followed by a skip statement pause. This is executed by three threads (see specification below), plus the main thread.

This debug session starts by putting a breakpoint on the skip pause loop, and then starting the *run* operation. The execution stops quickly on the skip in the MainThread, and the *threads* command indicates that there are three other threads, which have not started.

A sequence of *continue* commands allows each thread to run, eventually returning to the MainThread. This time the *threads* command shows that the other threads have all started and are stopped at line 9 (which is the skip loop).

The *stack* of the MainThread shows that the self object is A(#1). Using the *thread* command to switch to the Object-A-14 thread, a *stack* command shows that its self object is A(#4), and the skip loop is part way through when its time-slice ended, so x = 9989. A sequence of *continue* commands carries on around the threads until the execution is stopped with *stop* or *quit*. The result of interrupting the execution is usually *undefined* (since it did not really complete) and the execution time is the total duration of the debug session.

```
operations
      public op: () ==> ()
      op() ==
      (
            while true do
            (
                  IO`printf("Thread %s...\n", [threadid]);
                  for x = 1 to 10000 do skip
            )
      );

      public run: () ==> ()
      run() ==
      (
            startlist({new A(), new A(), new A()});
            op()
      );

thread
      op();
```

## 2.8.     Sessions

The state of a specification (ie. the *state* sections in a VDM-SL specification, or the static *instance variables* of VDM++ and VDM-RT specifications) is initialized once when the specification is loaded, and is modified thereafter by operation calls, for example using the *print <expression>* from the command prompt. The state can be restored to its initial state at any time using the *init* command.

For VDM++ and VDM-RT specifications, the state of any background threads, the CPUs they are allocated to, the BUSses that connect them, and the messages waiting to be sent between them are also maintained between command line calls. Though again, the *init* command can be used to clear all threads, CPUs, BUSses and messages, and reset the system to its initial state.

This leads to the concept of a *session*, where an interactive sequence of command line operations moves the model between states, just as though the user was a (very slow!) component in the system making operation calls. A session starts when the model is initialized (or re-initialized). Note that, while the command prompt is waiting for the user to type the next *print* command, the thread scheduling and normal BUS activity of the model is suspended - in effect, "time" in the simulation is stopped until the user types the next *print* expression, which allows the model to evolve until that expression has completed.

For example, consider the following VDM++ specification:

```
class A
instance variables
      iv:int := 0;

operations
      private inc: () ==> ()
      inc() == iv := iv + 1;

      public get: () ==> int
```

```
        get() == return iv;

        private run: () ==> ()
        run() == while true do inc();

        public static go: A ==> ()
        go(a) == start(a);

sync
        per inc => #fin(get) > #fin(inc);
        per get => #fin(get) = #fin(inc);

thread
        run();

end A
```

The model has a single instance variable, *iv*, which is incremented by the private *inc* operation, which is called in an infinite loop by the run operation, which in turn is the body of a thread. The thread is started by a static *go* operation. Permission predicates in the *sync* section mean that the *inc* operation can only be called once the *get* operation has "seen" each particular value of *iv*. So to test this model, we have to call *go*, and then make successive calls to *get. T*he background thread will continue running in between calls within the session. For example:

```
Parsed 1 class in 0.829 secs. No syntax errors
Type checked 1 class in 0.064 secs. No type errors
Initialized 1 class in 0.128 secs.
Interpreter started
> create a := new A()
> p a
= A{#1, iv:=0}
Executed in 0.0050 secs.
> threads
No threads running
> p a.get()
= 0
Executed in 0.012 secs.
> threads
No threads running
> p go(a)
= ()
Executed in 0.017 secs.
> threads
ObjectThread-10 (RUNNABLE)
> p a
= A{#1, iv:=1}
Executed in 0.0060 secs.
> p a.get()
= 1
Executed in 0.019 secs.
> threads
ObjectThread-10 (RUNNABLE)
> p a
= A{#1, iv:=2}
Executed in 0.0080 secs.
```

This illustrates a single session. Initially there are no threads running, as the *threads* command shows. The call to *go(a)* creates a thread for object *a* and successive calls to *a.get()* return the value of the *iv* state variable. The thread continues to run in the background between calls to *a.get()*.

## 2.9.    Interrupting Executions

Normally, the execution of a function or operation call will complete within a reasonable time and print the result on the console together with the execution time. However, sometimes executions can take a very long time. Such executions can be stopped by killing the entire process (typically CTRL-C, or a kill signal), but then it is not obvious where the evaluation was looping or why it was taking as long as it did.

Therefore VDMJ executions can be interrupted safely from the console. An interrupt can either pause the execution, such that it can be stepped and resumed, or it may stop the execution completely. A pause is similar to hitting a breakpoint; a stop is similar to raising a runtime exception.

If a *loop()* operation is defined as a simple *while true do skip* loop, the following illustrates an interrupted execution:

```
> p loop()
<hit return here>
Still executing: loop()
[p]ause to pause execution
[q]uit or [k]ill to cancel
pause<return>
Pausing...
12:         while true do
MainThread> s
13:             skip;
MainThread> s
12:         while true do
MainThread> s
13:             skip;
MainThread> c
k<return>
Terminating...
Error 4175: Execution cancelled in 'DEFAULT' (test.vdm) at line 12:15
MainThread> stack
Stopped [MainThread] in 'DEFAULT' (test.vdm) at line 12:15
In root context of loop() in 'DEFAULT' (console) at line 1:1
In root context of module scope
--
Error 4175: Execution cancelled in 'DEFAULT' (test.vdm) at line 13:9
```

Here, after the *loop()* operation is started, the user types <enter> before the execution completes (it never completes!). This causes the possible interrupt commands to be listed: either *pause*, which can be abbreviated to *p*, or *quit* or *kill*, which can be abbreviated as *q* or *k*, that stops the execution.

In the example, the user chooses *p*, which stops at line 12 as though a breakpoint had been reached. The session then steps the execution a few times before continuing. Then a *q* command is given, which causes an *Execution cancelled* exception to be raised. Finally, a *stack* command shows where the execution was interrupted.

The same technique can be used to interrupt the implicit initialization of a specification (or the execution of an *init* command), and it can also be used to interrupt *runtrace* commands.

## 2.10.    Standard Libraries

There are three standard libraries available to VDMJ: *IO, MATH* and *VDMUtil*.  These enable the interpreter to call out to standard library functions in Java, for example to read data from a file or to perform various mathematical functions. They are packaged into a jar called "stdlib-<version>.jar".

To use libraries, you must include the library jar on your classpath. Library jars include the VDM sources of the library, plus any supporting Java native code needed (see [10]).

Unfortunately, if you require more than one jar on the Java classpath, you can no longer start VDMJ

using the "-jar" option. Instead, you have to set the classpath using "-cp" and then name the main class, which for convenience is available in the default Java package as "VDMJ". See the example below. Alternatively, if you install the whole VDMJ distribution, you can use the *vdmj.sh* script to do this for you.

Note that if your project does not already contain the VDM source of the library, the sources of the library modules that you mention will be copied out of the library into a "lib" folder and associated with your session. The library files remain when VDMJ closes.

For example, the first example below uses the sine function from the MATH library, assuming the VDM-SL source is called *MATH.vdmsl*. The second example does the same thing via the *vdmj.sh* shell script.

```
$ java -cp vdmj-4.n.n.jar:stdlib-4.n.n.jar VDMJ -vdmsl -i MATH.vdmsl
Parsed 1 module in 0.092 secs. No syntax errors
Type checked 1 module in 0.246 secs. No type errors
Initialized 1 module in 0.196 secs. No init errors
Interpreter started
> files
lib/MATH.vdmsl

> p sin(MATH`pi/4)
= 0.7071067811865475
Executed in 0.005 secs.
>

$ vdmj.sh -vdmsl -q -i MATH.vdmsl
> p sin(1)
= 0.8414709848078965
Executed in 0.006 secs.
> files
lib/MATH.vdmsl
>
```

## 2.11.    Plugins

VDMJ's functionality can be extended by including *plugins*. A plugin is a Java jar that you add to the classpath, like a library. Plugins can add command line switches and add new commands to an interactive session.

Under the covers, all VDMJ functionality (such as the parsing and type checking of your specification) is provided by plugins, and the fundamental ones are built in. But you have the option to add new ones to extend the basic functionality of your system. You can list the plugins in your system using the *plugins* command.

There is a "quickcheck" plugin included with the VDMJ distribution, and this is added to the classpath if you use the *vdmj.sh* script. The plugin adds a *quickcheck* (or *qc*) command, which appears if you use *help.*

```
> help
...
quickcheck [-c <file>]|[-f <file>] [<PO numbers>] - lightweight PO
verification
...
> plugins
AST = com.fujitsu.vdmj.plugins.analyses.ASTPluginSL
TC = com.fujitsu.vdmj.plugins.analyses.TCPluginSL
IN = com.fujitsu.vdmj.plugins.analyses.INPluginSL
PO = com.fujitsu.vdmj.plugins.analyses.POPluginSL
QC = quickcheck.plugin.QuickCheckPlugin
>
```

# 3. VDMJUnit Testing

Section 2.7 describes the debugging of specifications using features built into the command line tools, either directly in VDMJ or indirectly using the VDMJC client. But once a specification is working, it is also possible to automatically execute batches of tests by using VDMJUnit, which adds VDM interpreter support to the Java JUnit 4 framework [12].

JUnit tests are written in Java, and are usually executed as part of an automated build of a larger system, or executed piecemeal in a development IDE. The JUnit framework allows Java classes and methods to be annotated to identify them as tests, and then provides the means to execute all of the tests in a give class or package, reporting any errors encountered.

The VDMJUnit package includes base classes that provide tests with the means to load a specification from file(s), (re)initialize them, and make test evaluations against the specification loaded, including tests which look for specific runtime errors, such as invariant failures.

Here is a minimal example:

```
public class SpecTest extends VDMJUnitTestPP
{
        @BeforeClass
        public static void start() throws Exception
        {
                setRelease(Release.VDM_10);
                readSpecification("test.vpp");
        }

        @Before
        public void setUp()
        {
                init();
        }

        @Test
        public void test() throws Exception
        {
                assertEquals(10, runInt("new A().f(9)"));
        }
}
```

Where the file "test.vpp" might contain the following VDM class definition:

```
class A
functions
    public f: nat -> nat
    f(i) == i + 1
    pre i < 10;
end A
```

Since this test is of a VDM++ class, the JUnit test class extends *VDMJUnitTestPP*; similar base classes are provided for VDM-SL and VDM-RT.

@BeforeClass is a standard JUnit 4 annotation which indicates that this static method is to be executed once before all tests in the class. This is used to execute the *setRelease* and *readSpecification* methods, inherited from *VDMJUnitTestPP*, which loads the file name(s) passed, and parses and type checks them using the *VDM10* language extensions. There must be no parse or type checking errors. The *readSpecification* method takes a varargs list of filenames or directories which are located relative to the Java classpath. Directories are searched (shallow) for all VDM source files. It is permissible to pass no files and evaluate bare expressions in the tests (like "1+1"), but the *readSpecification* method must always be called, and called only once per test class.

The @Before annotation is also from JUnit 4, and indicates that this method should be executed before each test defined in the class. The *init* method shown initializes the specification (eg. setting the state data back to the original settings). Without an @Before method to do this, tests will see the effect of earlier tests on the specification state (which may be useful). If the specification is not initialized per test, then it must be initialized once in the @BeforeClass method, after the specification is loaded.

Lastly, the @Test annotation identifies test methods, in this example a test asserting that a particular expression should produce a given (integer) value. The string argument to *runInt* can be any valid VDM expression that returns an integer value. The simpler *run* method returns a raw VDMJ *Value* object, which can return arbitrarily complex values (sets, records, objects etc).

A method called *assertVDM* is provided to make assertions about VDM values by using VDM expressions. This can be useful if it is too difficult to unpick the Value objects returned, when a VDM expression would be far simpler. The methods set a "RESULT" variable from the value passed in (or as a result of evaluating the expression passed), and this should be used to create a boolean VDM assertion expression. Assertion expressions can use global constants defined in the specification.

```java
@Test
public void one() throws Exception
{
        run("setValue(123)");
        assertVDM("getValue()", "RESULT = 123");

        try
        {
                Value r = run("getValue()");
                assertVDM("Testing!", r, "RESULT = 456");
                fail("Expected failure");
        }
        catch (AssertionError e)
        {
                assertEquals(e.getMessage(), "Testing!");
        }
}
```

Tests can catch and check exceptions if they believe a VDM runtime error should occur, for example:

```java
@Test
public void one() throws Exception
{
        try
        {
                create("object", "new A()");
                run("object.f(100)");
                fail("Expecting precondition failure");
        }
        catch (ContextException e)
        {
                // Error 4055: Precondition failure: pre_f in 'A' at line 5:11
                assertEquals(4055, e.number);
                assertEquals("A", e.location.module);
                assertEquals("test.vpp", e.location.file.getName());
                assertEquals(5, e.location.startLine);
                assertEquals(11, e.location.startPos);
        }
}
```

In this case, a VDMJ *ContextException* contains all of the information about the error that occurred, which can be checked by the test. Note that an *init* call should be made after an error to reset the specification to a known state, perhaps in the @Before method.

It is also possible to execute combinatorial tests from within the VDMJUnit framework (see section 4). As with the command line, all the tests generated from a trace may be executed, or a contiguous range of tests, or a reduced subset of the tests. Three *runTrace* methods are provided, as follows:

```java
@Test
public void one() throws Exception
{
        assertTrue(runTrace("T1"));
}

@Test
public void two() throws Exception
{
        assertTrue(runTrace("T1", 1, 3));
}

@Test
public void three() throws Exception
{
        assertTrue(runTrace("T1", 0.5, TraceReductionType.RANDOM, 123));
}
```

The first method executes all the tests generated from the "T1" trace. The second method runs a range of tests from those generated (tests are numbered from 1). The third method reduces the number of tests generated to 50% (ie. 0.5) using a RANDOM reduction technique, seeded with the value 123 (for reproducible results). There is a description of test reduction in section 4.2.

In all three cases, if all of the tests pass, the *runTrace* method returns true else false. Note that INCONCLUSIVE results are regarded as a failure. The expansion of the individual test cases, including test numbers and the output from each test execution, is sent to stdout in the same format as they are from the command line (see section 4).

The VDMJUnitTest framework provides the following methods to tests:

- **setRelease(Release)** to specify the language release to parse. The default is VDM classic.
- **readSpecification([Charset], file/dirs...)** to parse and type check specification files.
- **getErrors()** to get a List<VDMError> of syntax or type checking errors after readSpecification
- **getWarnings()** to get a List<VDMWarning> after readSpecification
- **init** to (re)initialize a loaded specification.
- **setDefault(Name)** to set the default module or class name.
- **create(Name, Expr)** for VDM++ and VDM-RT only, to create temporary object values
- **run(Expr)** to parse and evaluate the expression and return a VDMJ Value.
- **runInt(Expr)** same as run, but return a long.
- **runReal(Expr)** same as run, but return a double.
- **runBool(Expr)** same as run, but return a boolean.
- **runTrace(Name)** run all the tests from the named combinatorial trace.
- **runTrace(Name, first, last)** run the specific test range from a trace.
- **runTrace(Name, subset, method, seed)** run a subset of tests, reduced by the given method.
- **assertVDM([Message], Value, VDM assertion)** test value against a VDM assertion
- **assertVDM([Message], Expr, VDM assertion)** test VDM expression against VDM assertion

Note that a Java *Charset* can be passed to *readSpecification*, if the files are not in the default character encoding for your locale. See the Javadocs for full details.

A subclass of the VDMJ *Value* class is returned from the *run* method, which allows any value to be returned. Such *Value* objects either have fields or getter methods to allow them to be examined, though note that many *Value* subtypes contain other *Values* – for example, a *SetValue* contains a set of *Values*, and a *RecordValue* contains a map of String field names to *Values*. *Values* can be converted to Java primitives using (for example) the *boolValue* or *realValue* methods.

To execute JUnit tests with VDMJUnit, you need to have the VDMJ jar as well as the Junit 4 and VDMJUnit jars on the classpath.

Although *VDMJUnitTestSL* and related classes are intended to be used with JUnit tests, the same methods can be used in stand-alone Java applications. For example, the following code will load a specification and execute an expression passed from the command line.

```java
public class NonJUnitExample extends VDMJUnitTestSL
{
        public static void main(String[] args) throws Exception
        {
                new NonJUnitExample().execute(args);
        }

        public void execute(String[] args) throws Exception
        {
                setRelease(Release.VDM_10);
                readSpecification(args[0]);
                init();
                System.out.println(run(args[1]));
        }
}
```

# 4. Combinatorial Testing

(This section is a simplified version of [16]).

Creating a comprehensive set of tests for VDM specifications can be a very time consuming process. To try to make the generation of test cases simpler, VDMJ supports a VDM language extension (for all dialects) known as *Combinatorial Testing* [7].

The idea behind combinatorial testing is that classes or modules can be tested by making a sequence of calls on the interface of the class/module, but that the number of sensible sequences of operation calls is too great to be written out by hand, requiring automatic generation support.

To provide this, a class or module may define a "traces" section that defines an arbitrary number of trace specifications. Each of these is a *pattern* which expands into a number of operation call sequences to be made against one or more instances of the classes under test (which is usually not the class with the traces definition).

For example:

```
class Tested
instance variables
      total:int := 0;

operations
      public op1: () ==> int
      op1() == ( total := total + 1; return total; )

end Tested


class Tester
instance variables
      obj:Tested := new Tested();

traces
      test1: obj.op1();
      test2: obj.op1(); obj.op1(); obj.op1();

end Tester
```

This defines a class called Tester which is designed to test the class Tested. Notice that the Tester creates an instance of the object to be tested (obj) and that this object is referenced in the trace clauses, where a sequence of operation calls are specified. VDMJ provides a *runtrace* command to execute such traces.

If this specification is loaded into the VDMJ interpreter, the following output is produced:

```
> runtrace Tester`test1
Generated 1 tests
Test 1 = obj.op1()
Result = [1, PASSED]
Executed in 0.013 secs.
All tests passed

> runtrace Tester`test2
Generated 1 tests
Test 1 = obj.op1(); obj.op1(); obj.op1()
Result = [1, 2, 3, PASSED]
Executed in 0.01 secs.
All tests passed
```

Each test first prints out the sequences of calls they will make. Both traces only produce one test

---

sequence so they are both called "Test 1", but there can be several (see below). Then each test sequence is executed, printing out a sequence of results from the operation steps, followed by a *verdict* on how the test sequence completed.

Suppose, based on the tests above, that the Tested class was really designed to have the op1 operation called a number of times in sequence. We've tested one call, and three calls. But what about testing two, or a hundred and two, and everything in between? Rather than specifying all these test cases individually, combinatorial testing allows you to specify repetition patterns for operation calls. So, for example, if test2 is changed to

```
        test2: obj.op1(){1,5}
```

That indicates that the op1 call is to be made once, then twice, then three times and so on, up to five times – ie. this single trace specification actually describes five separate tests:

```
> runtrace Tester`test2a
Generated 5 tests
Test 1 = obj.op1()
Result = [1, PASSED]
Test 2 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
Test 3 = obj.op1(); obj.op1(); obj.op1()
Result = [1, 2, 3, PASSED]
Test 4 = obj.op1(); obj.op1(); obj.op1(); obj.op1()
Result = [1, 2, 3, 4, PASSED]
Test 5 = obj.op1(); obj.op1(); obj.op1(); obj.op1(); obj.op1()
Result = [1, 2, 3, 4, 5, PASSED]
Executed in 0.02 secs.
All tests passed
```

Notice that now there are five tests listed, Test 1 to Test 5, and that they each has one more op1 call than the previous test. Note also that the object being tested has been *initialized* between each test – since the results start at 1 every time.

Far more complicated test specifications can be given, involving several different pattern types, each of which expand into multiple tests. These are described in more detail in [7], but the following illustrates a combination of patterns:

```
        test3:
            let x in set {1,...,10} be st x mod 2 = 0 in
                let y = x + 1 in
                    (obj.op1(); obj.op2(x,y){1,2} | obj.op1())
```

When this is executed, the 'x' takes *all* the values 2, 4, 6, 8 and 10; for each value of x, y takes the value of x+1; and then the bracketed set of operations are called, with an op1 followed by *either* one and two calls to op2, *or* a single call to op1. So that's five values for x and y, with three alternative call sequences for each, making 15 test sequences in all. If op2 adds its two arguments to the running total, we get:

```
> runtrace Tester`test3
Generated 15 tests
Test 1 = obj.op1(); obj.op2(2, 3)
Result = [1, 6, PASSED]
Test 2 = obj.op1(); obj.op2(2, 3); obj.op2(2, 3)
Result = [1, 6, 11, PASSED]
Test 3 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
Test 4 = obj.op1(); obj.op2(4, 5)
Result = [1, 10, PASSED]
```

```
Test 5 = obj.op1(); obj.op2(4, 5); obj.op2(4, 5)
Result = [1, 10, 19, PASSED]
Test 6 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
Test 7 = obj.op1(); obj.op2(6, 7)
Result = [1, 14, PASSED]
Test 8 = obj.op1(); obj.op2(6, 7); obj.op2(6, 7)
Result = [1, 14, 27, PASSED]
Test 9 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
Test 10 = obj.op1(); obj.op2(8, 9)
Result = [1, 18, PASSED]
Test 11 = obj.op1(); obj.op2(8, 9); obj.op2(8, 9)
Result = [1, 18, 35, PASSED]
Test 12 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
Test 13 = obj.op1(); obj.op2(10, 11)
Result = [1, 22, PASSED]
Test 14 = obj.op1(); obj.op2(10, 11); obj.op2(10, 11)
Result = [1, 22, 43, PASSED]
Test 15 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
Executed in 0.055 secs.
All tests passed
```

If a sequence of operations causes a postcondition failure in a class, then it is certain that there is a problem with the specification – it should not be possible to provoke a post condition failure with a set of legal calls (ie. ones which pass the preconditions and type invariants). On the other hand, if a sequence of operations violates a precondition, or a type or class invariant, then it is *possible* that the specification has a problem, but it is also possible that the test itself is at fault (passing illegal values).

The combinatorial testing environment indicates the exit status of the test in the verdict returned in the last item of the results (all PASSED above). So if pre/post/invariant conditions are violated during a test, this may be set to FAILED or INDETERMINATE. If a test fails, then any subsequent test which starts with the same sequence of calls as the entire failed sequence will also fail. These tests are filtered out of the remaining test sequence automatically, and not executed.

For example, if we add a post condition to the op2 operation, saying that the running total must be less than 10, we get the following behaviour for test3:

```
class Tested
...
    public op2: int * int ==> int
    op2(x, y) == ( total := total + x + y; return total; )
    post total < 10;
...
```

**> runtrace Tester`test3**
```
Generated 15 tests
Test 1 = obj.op1(); obj.op2(2, 3)
Result = [1, 6, PASSED]
Test 2 = obj.op1(); obj.op2(2, 3); obj.op2(2, 3)
Result = [1, 6, Error 4072: Postcondition failure: post_op2 in 'Tested'
(test.vpp) at line 14:20, FAILED]
Test 3 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
Test 4 = obj.op1(); obj.op2(4, 5)
Result = [1, Error 4072: Postcondition failure: post_op2 in 'Tested'
(test.vpp) at line 14:20, FAILED]
Test 5 = obj.op1(); obj.op2(4, 5); obj.op2(4, 5)
Test 5 FILTERED by test 4
Test 6 = obj.op1(); obj.op1()
Result = [1, 2, PASSED]
```

```
...
```

Notice that the error message is included in the result sequence, and that tests 2 and 4 fail. Furthermore, the system knows that test 5 *would* fail as it starts with the same sequence that failed in test 4, so this is filtered from the run (it is listed, but not executed).

For large test runs, it can be useful to redirect the output to a file rather than let it scroll past on the screen. This can be done with the *savetrace* command.

```
> savetrace outputfile
runtrace output redirected to outputfile
> runtrace Tester`test3
Trace output sent to outputfile
Executed in 0.111 secs.
Some tests failed or indeterminate
> savetrace off
runtrace output is not redirected
>
```

For specifications with large numbers of trace definitions, it can be useful to run all of the traces, or all of the traces defined in a particular module or class, in one go. This can be done with the *runalltraces* command, which optionally takes a pattern argument to select module or class names.

```
> runalltraces
------------------------------------
runtrace A`T
Generated 1 tests in 0.019 secs.
Test 1 = op()
Result = [(), PASSED]
Executed in 0.0070 secs.
All tests passed
------------------------------------
runtrace B`T
Generated 1 tests in 0.0 secs.
Test 1 = op()
Result = [(), PASSED]
Executed in 0.0010 secs.
All tests passed
>
```

## 4.1.    Debugging Tests

When tests are producing failures, it is obviously desirable to be able to debug the tests as one would any other expression evaluation. Breakpoints set in the specification will be honoured when a combinatorial test sequence is expanded and executed (see 2.4), but by default the execution of a combinatorial test will run all of the tests that are expanded from the trace statement. To permit more precise debugging, it is possible to pass a single test number to the *runtrace* command, which will cause just one test to be executed – and therefore debugged in isolation. For example:

```
traces
    T1: let x in set {111, 222} in obj.op(x){1, 3}


> runtrace Tester`T1          -- run everything, as before
Generated 6 tests
Test 1 = obj.op(111)
Result = [111, PASSED]
Test 2 = obj.op(111); obj.op(111)
Result = [111, 222, PASSED]
Test 3 = obj.op(111); obj.op(111); obj.op(111)
Result = [111, 222, 333, PASSED]
```

```
Test 4 = obj.op(222)
Result = [222, PASSED]
Test 5 = obj.op(222); obj.op(222)
Result = [222, 444, PASSED]
Test 6 = obj.op(222); obj.op(222); obj.op(222)
Result = [222, 444, 666, PASSED]
Executed in 0.02 secs.
All tests passed
```

**> runtrace Tester`T1 5          -- just run test 5**
```
Generated 6 tests
Test 5 = obj.op(222); obj.op(222)
Result = [222, 444, PASSED]
Excluded 5 tests
Executed in 0.005 secs.
All tests passed
```

**> break op**
```
Created break [1] in 'Tested' (test.vpp) at line 10:18
10:        op(a) == ( total := total + a; return total; );
```
**> runtrace Tester`T1 5**
```
Generated 6 tests
Stopped break [1] in 'Tested' (test.vpp) at line 10:18
10:        op(a) == ( total := total + a; return total; );
```
**[CTMainThread]> p total**
```
total = 0
```
**[CTMainThread]> c**
```
Stopped break [1] in 'Tested' (test.vpp) at line 10:18
10:        op(a) == ( total := total + a; return total; );
```
**[CTMainThread]> p total**
```
total = 222
```
**[CTMainThread]> p a**
```
a = 222
```
**[CTMainThread]> c**
```
Test 5 = obj.op(222); obj.op(222)
Result = [222, 444, PASSED]
Excluded 5 tests
Executed in 41.841 secs.
All tests passed
```

**> runtrace Tester`T1 8**
```
Error: Trace Tester`T1 only has 6 tests
```

Unlike the regular runtime, when a set of combinatorial tests are being executed, any exceptions (for example postcondition failures) are regarded as the "final result" of one test and evaluation moves onto the next test in the set. Normally an exception would halt execution and trap into the debugger. This means that although you can set breakpoints and step through a test as above, you cannot simply let a complex test run and expect it to trap into the debugger when it hits an exception.

For example, consider the following simple test. This will produce a division by zero error when it is executed, and that becomes the result of the test rather than halting the execution.

```
class A
operations
    public op: int * int ==> ()
    op(a, b) == let - = a/b in skip;

end A

class B
instance variables
    obj:A := new A();
```

```
traces
      T: let x in set {-1, 0, 1} in obj.op(1, x);

end B
```

If we simply run the B`T test, this will produce a result with a divide by zero error in the 2nd test:

```
> runtrace B`T
Generated 3 tests
Test 1 = obj.op(1, -1)
Result = [(), PASSED]
Test 2 = obj.op(1, 0)
Result = [Error 4134: Infinite or NaN trouble in 'A' (test.vpp) at line
4:26, FAILED]
Test 3 = obj.op(1, 1)
Result = [(), PASSED]
Executed in 0.014 secs.
Some tests failed or indeterminate
>
```

So we might try to set a breakpoint on the operation, re-run just the second test and step into the evaluation.

```
> break op
Created break [1] in 'A' (test.vpp) at line 4:17
4:     op(a, b) == let - = a/b in skip;
> runtrace B`T 2
Generated 3 tests
Stopped break [1] in 'A' (test.vpp) at line 4:17
4:     op(a, b) == let - = a/b in skip;
[CTMainThread]> n
Test 2 = obj.op(1, 0)
Result = [Error 4134: Infinite or NaN trouble in 'A' (test.vpp) at line
4:26, FAILED]
Excluded 2 tests
Executed in 10.305 secs.
Some tests failed or indeterminate
>
```

Unfortunately, because the exception is treated as the final result of the test, this does not trap into the debugger when the exception occurs. Instead, the exception string is simply recorded with the previous step return values and given as the overall result of the test. So to run combinatorial tests in a mode where exceptions *do* trap into the debugger, you can start them with a different command: *debugtrace*. In this mode, we don't need the breakpoint to let us "step up to" the error, we can just let it run until it fails, and traps into the debugger.

```
> remove 1
Cleared break [1] in 'A' (CT.vpp) at line 4:17
4:     op(a, b) == let - = a/b in skip;
> debugtrace B`T 2
Generated 3 tests
Stopped in 'A' (test.vpp) at line 4:26
4:     op(a, b) == let - = a/b in skip;
[CTMainThread]> p a
a = 1
[CTMainThread]> p b
b = 0
[CTMainThread]> stack
Stopped [CTMainThread] in 'A' (test.vpp) at line 4:26
In context of let statement in 'A' (test.vpp) at line 4:17
      self = A{#1}
```

```
        b = 0
        a = 1
In object context of op(a, b) in 'B' (test.vpp) at line 13:35
In object context of B() in 'B' (test.vpp) at line 8:7
In root context of public static environment
[CTMainThread]> c
Test 2 = obj.op(1, 0)
Result = [Error 4134: Infinite or NaN trouble in 'A' (test.vpp) at line
4:26, FAILED]
Excluded 2 tests
Executed in 49.197 secs.
Some tests failed or indeterminate
>
```

## 4.2.    Filtering Tests

Part of the power of the combinatorial testing feature is that it can automatically generate and execute very large sets of tests. On the other hand, very large sets of tests can take a very long time to execute, so it is sometimes desirable to reduce the number of tests generated in the early stages of testing. Problems can then be corrected and the reduced set of tests can be re-executed quickly to test the fixes before the full set of tests are executed.

Consider the following trace definition:

```
traces
      T: let x in set {1, ..., 20} in op(x){1,6};
```

That generates 120 tests...

```
> runtrace T
Generated 120 tests
Test 1 = op(1)
Result = [(), PASSED]
Test 2 = op(1); op(1)
Result = [(), (), PASSED]
Test 3 = op(1); op(1); op(1)
Result = [(), (), (), PASSED]
Test 4 = op(1); op(1); op(1); op(1)
Result = [(), (), (), (), PASSED]
...
Test 117 = op(20); op(20); op(20)
Result = [(), (), (), PASSED]
Test 118 = op(20); op(20); op(20); op(20)
Result = [(), (), (), (), PASSED]
Test 119 = op(20); op(20); op(20); op(20); op(20)
Result = [(), (), (), (), (), PASSED]
Test 120 = op(20); op(20); op(20); op(20); op(20); op(20)
Result = [(), (), (), (), (), (), PASSED]
Executed in 0.163 secs.
All tests passed
>
```

You can see how a similar trace could generate thousands or millions of tests. The simplest way to reduce the number of tests executed is to specify a range of test numbers to run. For example:

```
> runtrace T 100 110
Generated 120 tests
Test 100 = op(17); op(17); op(17); op(17)
Result = [18, 18, 18, 18, PASSED]
Test 101 = op(17); op(17); op(17); op(17); op(17)
Result = [18, 18, 18, 18, 18, PASSED]
Test 102 = op(17); op(17); op(17); op(17); op(17); op(17)
```

```
Result = [18, 18, 18, 18, 18, 18, PASSED]
Test 103 = op(18)
Result = [19, PASSED]
Test 104 = op(18); op(18)
Result = [19, 19, PASSED]
Test 105 = op(18); op(18); op(18)
Result = [19, 19, 19, PASSED]
Test 106 = op(18); op(18); op(18); op(18)
Result = [19, 19, 19, 19, PASSED]
Test 107 = op(18); op(18); op(18); op(18); op(18)
Result = [19, 19, 19, 19, 19, PASSED]
Test 108 = op(18); op(18); op(18); op(18); op(18); op(18)
Result = [19, 19, 19, 19, 19, 19, PASSED]
Test 109 = op(19)
Result = [20, PASSED]
Test 110 = op(19); op(19)
Result = [20, 20, PASSED]
Excluded 109 tests
Executed in 0.012 secs.
All tests passed
>
```

Notice that the tests executed include the range specified, and that as a result 109 of the 120 tests were excluded. This method of trace reduction is useful if a very large number of tests need to be executed, and it is convenient to (say) execute one million per day, and check the results.

Another way to reduce the number of tests executed is the *filter* command. This is a global setting that applies to all combinatorial tests. The simplest way to use the command is to specify a percentage of the tests you want to run – say 10%.

```
> filter 10
Trace filter currently 10.0% RANDOM (seed 0)
> runtrace T
Generated 120 tests, reduced to 12
Test 5 = op(1); op(1); op(1); op(1); op(1)
Result = [(), (), (), (), (), PASSED]
Test 41 = op(7); op(7); op(7); op(7); op(7)
Result = [(), (), (), (), (), PASSED]
Test 43 = op(8)
Result = [(), PASSED]
Test 56 = op(10); op(10)
Result = [(), (), PASSED]
Test 62 = op(11); op(11)
Result = [(), (), PASSED]
Test 70 = op(12); op(12); op(12); op(12)
Result = [(), (), (), (), PASSED]
Test 71 = op(12); op(12); op(12); op(12); op(12)
Result = [(), (), (), (), (), PASSED]
Test 78 = op(13); op(13); op(13); op(13); op(13); op(13)
Result = [(), (), (), (), (), (), PASSED]
Test 82 = op(14); op(14); op(14); op(14)
Result = [(), (), (), (), PASSED]
Test 92 = op(16); op(16)
Result = [(), (), PASSED]
Test 103 = op(18)
Result = [(), PASSED]
Test 109 = op(19)
Result = [(), PASSED]
Excluded 108 tests
Executed in 0.039 secs.
All tests passed
>
```

Notice that now we only generate 12 tests (10% of 120). The 12 we get are selected at random from

---

the 120, so you can see the first one is number 5, then 41, and the last two are 103 and 109. If you run the test again *you will get precisely the same selection of tests*. This is important, because you want the selection to be repeatable while you are testing. The *seed* for the random number generator can be changed using the *seedtrace* command:

```
> seedtrace 12345
Seed now set to 12345
> filter 10
Trace filter currently 10% RANDOM (seed 12345)
>
```

A random selection of tests is a simple way to reduce the set, but it runs the risk of eliminating entire "classes" of tests that are trying to check particular behaviour. Therefore the filtering has several other ways to reduce the number of tests, in addition to the default random method. These are based on the "shapes" of the tests, which considers the sequence of the names of the operations and functions called. By preserving at least one test of every shape in the full test set, the reduction can give a more representative result (though it may not be able to achieve the reduction percentage requested).

```
> filter shapes_novars
Trace filter currently 10% SHAPES_NOVARS (seed 12345)
> runtrace T
Generated 120 tests
Test 1 = op(1)
Result = [(), PASSED]
Test 2 = op(1); op(1)
Result = [(), (), PASSED]
Test 3 = op(1); op(1); op(1)
Result = [(), (), (), PASSED]
Test 4 = op(1); op(1); op(1); op(1)
Result = [(), (), (), (), PASSED]
Test 5 = op(1); op(1); op(1); op(1); op(1)
Result = [(), (), (), (), (), PASSED]
Test 6 = op(1); op(1); op(1); op(1); op(1); op(1)
Result = [(), (), (), (), (), (), PASSED]
Test 71 = op(12); op(12); op(12); op(12); op(12)
Result = [(), (), (), (), (), PASSED]
Test 78 = op(13); op(13); op(13); op(13); op(13); op(13)
Result = [(), (), (), (), (), (), PASSED]
Test 82 = op(14); op(14); op(14); op(14)
Result = [(), (), (), (), PASSED]
Test 92 = op(16); op(16)
Result = [(), (), PASSED]
Test 103 = op(18)
Result = [(), PASSED]
Test 109 = op(19)
Result = [(), PASSED]
Excluded 108 tests
Executed in 0.045 secs.
All tests passed

> filter
Usage: filter %age | RANDOM | SHAPES_NOVARS | SHAPES_VARNAMES |
SHAPES_VARVALUES
Trace filter currently 10% SHAPES_NOVARS (seed 0)
```

Notice that in the random 10% reduction (giving 12 tests) no tests were selected that had three op calls. There are tests with three op calls, but this "shape" is missed entirely. By changing the filter to use shapes, a different set of tests is selected and this time those with all shapes are represented. The first new instance of every shape will be present in the selected tests, plus a random selection of tests to achieve the filter percentage reduction requested.

The other reduction options additionally consider "let" variable names and their values when identifying shapes. See [11].

# 5. Annotations

## 5.1. Overview

Annotations were introduced in VDMJ version 4.3.0 as a means to allow a specifier to affect the tool's behaviour without affecting the meaning of the specification. The idea is similar to the notion of annotations in Java, which can be used to affect the Java compiler, but do not alter the runtime behaviour of a program.

VDMJ provides some standard annotations, but the intent is that specifiers can create new annotations and add them to the VDMJ system easily.

Full details of VDMJ annotation support is in the Annotation Guide [14].

## 5.2. Syntax

Annotations are added to a specification as comments, either block comments or one-line comments. This is so that other VDM tools will not be affected by the addition of annotations, and emphasises the idea that annotations do not alter the meaning of a specification.

An annotation must be present at the start of a comment, and has the following default syntax:

```
'@', identifier, [ '(', expression list, ')' ]
```

So for example, an operation in a VDM++ class could be annotated as follows:

```
class A
operations
    -- @Override
    public add: nat * nat ==> nat
    add(a, b) == ...
```

Or the value of variables can be traced during execution as follows:

```
functions
    add: nat * nat +> nat
    add(a, b) ==
        /* @Trace(a, b) */ a + b;
```

# 6. Internationalization (I18N)

Often, a VDM specification will simply be written and executed in the default locale, and the character set and input/output methods of the user's editor and VDMJ's parser and interactive console will work together naturally.

However, sometimes a specification must be included that has been written in a different locale. VDM keywords always use the Latin character set, but user defined names and strings may be localized (eg. in Greek, Japanese or Cyrillic, or there may be currency symbols or accented characters in the specification). This means that the character set and encoding used in the specification file must be understood by the VDMJ parser, and the console (in the interpreter) must be able to display the specification's characters.

For example, a specification may be written in Japanese and saved to a file using "Shift JIS" encoding. If this is parsed with VDMJ's default options with a UK locale, the following error is likely:

```
$ java -jar vdmj-4.n.n.jar -vdmpp -w bankaccount.vpp
Error 1009: Unexpected character '?' (code 0x2039) in 'bankaccount.vpp'
(bankaccount.vpp) at line 1:8
Parsed 0 classes in 0.031 secs. Found 1 syntax error
```

Here the parser has failed fairly early on the first line, probably shortly after the keyword "class" where the Japanese name of a class appears. Because we know that the file is Shift JIS encoded, we can tell the parser to read it using the option "-c SJIS":

```
$ java -jar vdmj-4.n.n.jar -vdmpp -w -c SJIS bankaccount.vpp
Parsed 1 class in 0.39 secs. No syntax errors
Type checked in 0.032 secs. No type errors and suppressed 1 warning
```

Now the Japanese characters have been read correctly, and turned into Java's internal representation for all characters (Unicode). But there is still a problem if this specification is interpreted, because the names cannot be displayed in the console (which uses the default locale still):

```
$ java -jar vdmj-4.n.n.jar -vdmpp -w -i -c SJIS bankaccount.vpp
Parsed 1 class in 0.219 secs. No syntax errors
Type checked in 0.031 secs. No type errors and suppressed 1 warning
Initialized 1 class in 0.0 secs.
Interpreter started
> env
????`inv_??(????`??) = (????`?? +> bool)
>
```

Here the environment cannot be displayed because the name of the function involved is not displayable in the default locale. There is a separate option to set the console's character set, -t. For example, if the console is UTF8 (Unicode), this would be set as follows:

```
$ java -jar vdmj-4.n.n.jar -vdmpp -i -c SJIS -t UTF8 bankaccount.vpp
Parsed 1 class in 0.235 secs. No syntax errors
Warning 5001: Instance variable is not initialized: 銀行口座`所有者 in '銀行口
座' (bankaccount.vpp) at line 10:5
Type checked in 0.047 secs. No type errors and 1 warning
Initialized 1 class in 0.0 secs.
Interpreter started
> env
銀行口座`inv_数字(銀行口座`数字) = (銀行口座`数字 +> bool)
>
```

Appendix B lists the character set names that may be used with -c or -t.

---

# 7.     The vdmj.properties File

The command line options described in section  2.2 could typically be different from run to run. But some settings are so constant that they are more conveniently held in an optional properties file. By creating this file and changing the settings, all subsequent executions of VDMJ will be affected.

The properties file is called "vdmj.properties" and must be in a directory on the Java classpath (eg. the same directory as the VDMJ jar).  If the file is not present, or an entry is missing from the file, then the default value is used (shown for each name below).

The property names are all of the form `vdmj.<area>.<some_name>`.

The default vdmj.properties file is as follows:

```
#
# Settings for VDMJ. These override defaults in the code.
#

# The tab stop for source files (default 4)
vdmj.parser.tabstop = 4
```

The tabstop is used by the parser and error reporting system to accurately report location information for syntax and runtime errors. If this value does not match the tabstop set in the editor you use, then the column number for errors will be incorrect on lines with tabs.

```
# Nesting of block comments: 0=support, 1=warning, 2=error, 3=ignore
vdmj.parser.comment_nesting = 3
```

This property controls the behaviour of nested block comments. They are either supported, supported but giving a warning or error, or ignored.

```
# Enable T! maximal type parsing
vdmj.parser.maximal_types = false
```

This property allows types to be declared as T! or mk_R!, indicating that they are *maximal* types that suppress the invariants/equality/order clauses of T and R, respectively.

```
# The package list for annotation classes to load.
vdmj.annotations.packages = com.fujitsu.vdmj.ast.annotations;annotations.ast
```

This is a list of packages that is used to look for annotation classes. For example @Warning would look for a class called ASTWarningAnnotation within these packages.

```
# Enable annotation debugging (default false)
vdmj.annotations.debug = false
```

This flag enables some debugging output for annotations.

```
# An alternative search path for the ClassMapper (default null)
# vdmj.mapping.search_path = null
```

This property gives the ClassMapper locations to search for mapping files, in addition to the classpath.

```
# Skip the check for mutually-recursive function calls (default false)
vdmj.tc.skip_recursive_check = false
```

If set, this property prevents the check for mutually recursive function calls (which can be expensive).

```
# Skip the check for definition dependency cycles (default false)
```

```
vdmj.tc.skip_cyclic_check = false
```

If set, this property prevents the check for dependency cycles between definitions (which can be expensive).

```
# The maximum TC errors reported before "Too many errors" (default 100)
vdmj.tc.max_errors = 100
```

This is the maximum number of type checking errors that will be reported.

```
# The maximum number of POG paths in an operation (default 200)
vdmj.pog_max_alt_paths = 200
```

The proof obligation generator for operations has to calculate the possible paths through the body of the operation. This can quickly grow beyond a reasonable value, and is limited by this property.

```
# The default timeslice (statements executed) for the FCFS policy
vdmj.scheduler.fcfs_timeslice = 10
```

With VDM-RT, the first-come first-served (FCFS) scheduling policy allows every thread to run for a fixed timeslice before scheduling the next one. A timeslice is defined in terms of statements executed, so this property means that the default FCFS timeslice is run 10 statements or expressions.

```
# The vCPU/vBUS timeslice (default 10000)
vdmj.scheduler.virtual_timeslice = 10000
```

Virtual CPUs need to do as much work as possible in a timeslice, as they are intended to feed the rest of the system with events. Therefore they have a separate timeslice, which is larger than the default.

```
# The timeslice variation (+/- jitter ticks, default 0)
vdmj.scheduler.jitter = 0
```

To avoid perfectly deterministic execution (and so reveal synchronization problems that worked "by luck") it is possible to add a random amount of jitter to each timeslice. By default this is zero, which produces deterministic execution.

```
# The default duration for RT statements (default 2)
vdmj.rt.duration_default = 2
```

All statements in a VDM-RT specification have a duration, so time is guaranteed to move forward by this amount after ever statement is executed (unless a surrounding duration or cycles statement prevents it). This property defines the default duration. This may eventually be extended to allow different durations for different statement types, such as "rt.duration.assignment = 5".

```
# Enable transactional variable updates (default false)
vdmj.rt.duration_transactions = false
```

VDM-RT has the concept of transaction variable updates, where changes made to a variable in a thread are only visible to that thread until a timestep is completed. This is problematic for many specifications, and is disabled by default. This property enables it.

```
# Enable InstVarChange RT log entries (default false)
vdmj.rt.log_instvarchanges = false
```

VDM-RT can optionally log instance variable changes to the log file. By default this is not done, as it adds to the size of the log files, but it is useful to enable this when debugging.

```
# Enable extra diagnostics for guards etc.
```

```
# (default false)
vdmj.rt.diags_guards = false
```

Debugging a multi-threaded specification can be extremely difficult. This property turns on extra diagnostics (sent to stderr for VDM++, or the RT log file for VDM-RT) which indicate when guards are tested, block or pass.

```
# Enable extra RT log diagnostics for timesteps (default false)
vdmj.rt.diags_timestep = false
```

Debugging VDM-RT specifications can similarly be difficult from a timing point of view. This property turns on extra diagnostics, that are sent to the RT log, indicating when each thread is waiting to move time and by how much.

```
# The maximum number of expansions for "+" and "*" patterns (default 5)
vdmj.traces.max_repeats = 5
```

Combinatorial testing has "+" and "*" repeat patterns which produce an arbitrary number of iterations of the statement(s) being repeated. In practice, the tool has to decide how many iterations to make. This property defines the number.

```
# Serialize the system state between trace tests (default false)
vdmj.traces.save_state = false
```

If set, this property causes the trace execution system to serialise and restore the specification state between tests. This can save time if the state is very complex to re-calculate each time.

```
# The size below which trace function args are expanded (default 50)
vdmj.traces.max_arg_length = 50
```

Normally, trace execution tries to include the value of arguments passed to operations and functions, like "op(123)" rather than "op(a)". But if the expanded value is longer than this property limit, the symbolic name will be used.

```
# A list of user defined VDMJ plugins
# vdmj.plugins = <fully qualified class CSV list>
```

This option allows new analysis plugins to be added to VDMJ dynamically. These can define their own command line options, and console commands, as well as performing new analysis checks.

```
# The priority of plugins, affecting the order of event processing.
# Defaults are AST=100, TC=200, IN=300, PO=400, USER=1000
vdmj.plugin.priority.ast = 100
vdmj.plugin.priority.tc = 200
vdmj.plugin.priority.in = 300
vdmj.plugin.priority.po = 400
vdmj.plugin.priority.user = 1000
```

These properties can be used to change the priority of the built-in plugins to allow user plugins to insert themselves in different places.

```
# The class name for the DebugLink (default null)
# vdmj.debug.link_class = null
```

This allows the debugger to link to an alternative instance of the DebugLink class. It is used for connections to external IDEs.

```
# The set size limit for power set expressions */
vdmj.in.powerset_limit = 30
```

This setting causes the interpreter to fail with a meaningful error if it attempts to calculate the power set of a set containing more than this number of elements. This "fast fail" is usually better than allowing the interpreter to try to calculate billions of subsets, and eventually fail in an unhelpful way by running out of memory.

```
# The type size limit for type bind expansions
vdmj.in.typebind_limit = 100000
```

This setting is similar to the power set limit above, but it applies when expanding type binds for finite types that have more than a certain number of values. For example, *set of bool* has four values; but a *set of set of set of bool* has 65536 values. This can quickly explode.

```
# The maximum stack to dump via println(Throwable) */
vdmj.diag.max_stack = 1;
```

This is the maximum number of lines of a Java stack dump that will be printed.

# 8.      Appendix A: The shmem Example

The following is the source of the *shmem.vdm* example used in section  2.6.  It models the behaviour of the 32-bit shared memory quadrants of HP-UX, using a record type M to represent a block of memory which is either <FREE> or <USED>, and a sequence of M records to represent a Quadrant. The specification output indicates which allocation policy, first-fit or best-fit (or neither), produces the most memory fragmentation.

```
> p main(5,100)
= [<FIRST>, <SAME>, <FIRST>, <BEST>, <FIRST>]
Executed in 19.73 secs.


module M
exports all
definitions
types

Quadrant = seq of M;
--inv Q ==
--  forall a in set elems Q &
--     (not exists b in set elems Q \ {a} &
--             (b.start >= a.start and b.start <= a.stop) or
--             (b.stop  >= a.start and b.stop  <= a.stop));

M :: type : <USED> | <FREE>
     start: nat
     stop : nat
inv mk_M(-, a, b) == (b >= a)


state Memory of
  rseed: nat
  Q3: Quadrant
  Q4: Quadrant
inv mk_Memory(-, q3, q4) == len q3 > 0 and len q4 > 0
init q ==
  q = mk_Memory(87654321,
                [mk_M(<FREE>, 0, MAXMEM-1)],
                [mk_M(<FREE>, 0, MAXMEM-1)])

end

values

MAXMEM = 10000;
CHUNK = 100;

functions

sizeof: M -> nat1
sizeof(m) ==
  m.stop - m.start + 1;

least: nat1 * nat1 -> nat1
least(a, b) ==
  if a < b
  then a
  else b;

spacefor: nat1 * Quadrant -> nat1
spacefor(size, Q) ==
  cases Q:
    []          -> MAXMEM + 1,
```

```
                [h] ^ tail -> if h.type = <FREE> and sizeof(h) >= size
                              then sizeof(h)
                              else spacefor(size, tail)
    end
measure QuadrantLen;

QuadrantLen:  nat1 * Quadrant -> nat
QuadrantLen(-,list) ==
  len list;

bestfit: nat1 * Quadrant -> nat1
bestfit(size, Q) ==
  cases Q:
    -- as we're looking for the smallest
    []         -> MAXMEM + 1,
    [h] ^ tail -> if h.type = <FREE> and sizeof(h) >= size
                  then least(sizeof(h), bestfit(size, tail))
                  else bestfit(size, tail)
  end
measure QuadrantLen;

add: nat1 * nat1 * Quadrant -> Quadrant
add(size, hole, Q) ==
  cases Q:
    [h] ^ tail ->  if h.type = <FREE> and sizeof(h) = hole
                   then if hole = size
                        then [mk_M(<USED>, h.start, h.stop)] ^ tail
                        else [mk_M(<USED>, h.start, h.start + size - 1),
                              mk_M(<FREE>, h.start + size, h.stop)] ^ tail
                   else [h] ^ add(size, hole, tail),
    others -> Q
  end
pre hole >= size
measure QuadrantLen2;

QuadrantLen2: nat1 * nat1 * Quadrant -> nat
QuadrantLen2(-,-,list) ==
  len list;

combine: Quadrant -> Quadrant
combine(Q) ==
  cases Q:
    [h1, h2] ^ tail ->
       if h1.type = <FREE> and h2.type = <FREE>
       then combine([mk_M(<FREE>, h1.start, h2.stop)] ^ tail)
       else [h1] ^ combine(tl Q),
    others -> Q
  end
measure QuadrantLen0;

QuadrantLen0: Quadrant -> nat
QuadrantLen0(list) ==
  len list;

delete: M * Quadrant -> Quadrant
delete(item, Q) ==
  if hd Q = item
  then combine([mk_M(<FREE>, item.start, item.stop)] ^ tl Q)
  else [hd Q] ^ delete(item, tl Q)
  measure MQuadrantLen;

MQuadrantLen: M * Quadrant -> nat
MQuadrantLen(-,list) ==
  len list;
```

```
fragments: Quadrant -> nat
fragments(Q) ==
  card {x | x in set elems Q & x.type = <FREE>} - 1;


operations

seed: nat1 ==> ()
seed(n) ==
  rseed := n;

inc: () ==> ()
inc() ==
  for i = 1 to rseed mod 7 + 3
  do
    rseed := (rseed * 69069 + 5) mod 4294967296;

rand: nat1 ==> nat1
rand(n) ==
 (inc();
  return rseed mod n + 1;
 );

FirstFit: nat1 ==> bool
FirstFit(size) ==
 (let q4 = spacefor(size, Q4)
  in
    if q4 <= MAXMEM
    then Q4 := add(size, q4, Q4)
    else let q3 = spacefor(size, Q3)
         in
           if q3 <= MAXMEM
           then Q3 := add(size, q3, Q3)
           else return false;
  return true;
 );

BestFit: nat1 ==> bool
BestFit(size) ==
 (let q4 = bestfit(size, Q4)
  in
    if q4 <= MAXMEM
    then Q4 := add(size, q4, Q4)
    else let q3 = bestfit(size, Q3)
         in
           if q3 <= MAXMEM
           then Q3 := add(size, q3, Q3)
           else return false;
  return true;
 );

Reset: () ==> ()
Reset() ==
 (Q3 := [mk_M(<FREE>, 0, MAXMEM-1)];
  Q4 := [mk_M(<FREE>, 0, MAXMEM-1)];
 );

DeleteOne: () ==> ()
DeleteOne() ==
 (if rand(2) = 1
  then let i = rand(len Q3)
       in
          if Q3(i).type = <USED>
          then Q3 := delete(Q3(i), Q3)
          else DeleteOne()
  else let i = rand(len Q4)
```

```
          in
             if Q4(i).type = <USED>
             then Q4 := delete(Q4(i), Q4)
             else DeleteOne()
  )
pre (exists m in set elems Q3 & m.type = <USED>) or
    (exists m in set elems Q4 & m.type = <USED>);

TryFirst: nat ==> nat
TryFirst(loops) ==
 (dcl count:int := 0;

  while count < loops and FirstFit(rand(CHUNK))
  do
    (if count > 50
     then DeleteOne();
     count := count + 1;
    );

  -- return count;
  return fragments(Q3) + fragments(Q4);
 );

TryBest: nat ==> nat
TryBest(loops) ==
 (dcl count:int := 0;

  while count < loops and BestFit(rand(CHUNK))
  do
    (if count > 50
     then DeleteOne();
     count := count + 1;
    );

  -- return count;
  return fragments(Q3) + fragments(Q4);
 );

main: nat1 * nat1 ==> seq of (<BEST> | <FIRST> | <SAME>)
main(tries, loops) ==
 (dcl result: seq of (<BEST> | <FIRST> | <SAME>) := [];

  for i = 1 to tries
  do
    (dcl best:int, first:int;

     Reset();
     seed(i);
     best := TryBest(loops);

     Reset();
     seed(i);
     first := TryFirst(loops);

     if best = first
     then result := result ^ [<SAME>]
     elseif best > first
     then result := result ^ [<BEST>]
     else result := result ^ [<FIRST>];
    );

  return result;
 )

end M
```

# 9.    Appendix B: Supported Character Sets

The following character set names are supported by the -c and -t command line options by default. Extra charsets can be added by including the lib/charsets.jar file in the JRE. See http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html for more information.

The values in [braces] are aliases.  The list is printed whenever an unknown character set name is passed as an argument:

```
IBM00858 [cp858, ccsid00858, 858, cp00858]
IBM437 [ibm-437, windows-437, cspc8codepage437, 437, ibm437, cp437]
IBM775 [ibm-775, cp775, ibm775, 775]
IBM850 [ibm-850, cp850, 850, cspc850multilingual, ibm850]
IBM852 [ibm852, csPCp852, 852, ibm-852, cp852]
IBM855 [cspcp855, 855, ibm855, ibm-855, cp855]
IBM857 [csIBM857, 857, ibm-857, cp857, ibm857]
IBM862 [ibm-862, ibm862, csIBM862, cp862, cspc862latinhebrew, 862]
IBM866 [866, ibm-866, ibm866, csIBM866, cp866]
ISO-8859-1 [csISOLatin1, IBM-819, iso-ir-100, 8859_1, ISO_8859-1, l1,
ISO8859-1, ISO_8859_1, cp819, ISO8859_1, latin1, ISO_8859-1:1987, 819,
IBM819]
ISO-8859-13 [8859_13, iso8859_13, iso_8859-13, ISO8859-13]
ISO-8859-15 [IBM923, 8859_15, ISO_8859-15, ISO-8859-15, L9, ISO8859-15,
ISO8859_15_FDIS, 923, LATIN0, csISOlatin9, LATIN9, csISOlatin0, IBM-923,
ISO8859_15, cp923]
ISO-8859-2 [iso-ir-101, csISOLatin2, ibm-912, 8859_2, l2, ISO_8859-2,
ibm912, 912, ISO8859-2, latin2, iso8859_2, ISO_8859-2:1987, cp912]
ISO-8859-4 [iso-ir-110, iso8859-4, ibm914, ibm-914, l4, csISOLatin4, 914,
8859_4, latin4, ISO_8859-4, ISO_8859-4:1988, iso8859_4, cp914]
ISO-8859-5 [cp915, ISO8859-5, ibm915, ISO_8859-5:1988, ibm-915, 8859_5,
915, cyrillic, iso8859_5, ISO_8859-5, iso-ir-144, csISOLatinCyrillic]
ISO-8859-7 [iso8859-7, sun_eu_greek, csISOLatinGreek, 813, ISO_8859-7,
ISO_8859-7:1987, ibm-813, greek, greek8, iso8859_7, ECMA-118, iso-ir-126,
8859_7, cp813, ibm813, ELOT_928]
ISO-8859-9 [ISO_8859-9, 920, iso8859_9, csISOLatin5, l5, 8859_9, latin5,
ibm920, iso-ir-148, ISO_8859-9:1989, ISO8859-9, cp920, ibm-920]
KOI8-R [cskoi8r, koi8_r, koi8]
KOI8-U [koi8_u]
US-ASCII [cp367, ascii7, ISO646-US, 646, csASCII, us, iso_646.irv:1983,
ISO_646.irv:1991, IBM367, ASCII, default, ANSI_X3.4-1986, ANSI_X3.4-1968,
iso-ir-6]
UTF-16 [utf16, UTF_16, UnicodeBig, unicode]
UTF-16BE [X-UTF-16BE, UTF_16BE, ISO-10646-UCS-2, UnicodeBigUnmarked]
UTF-16LE [UnicodeLittleUnmarked, UTF_16LE, X-UTF-16LE]
UTF-32 [UTF_32, UTF32]
UTF-32BE [X-UTF-32BE, UTF_32BE]
UTF-32LE [X-UTF-32LE, UTF_32LE]
UTF-8 [UTF8, unicode-1-1-utf-8]
windows-1250 [cp1250, cp5346]
windows-1251 [ansi-1251, cp5347, cp1251]
windows-1252 [cp1252, cp5348]
windows-1253 [cp1253, cp5349]
windows-1254 [cp1254, cp5350]
windows-1257 [cp1257, cp5353]
x-IBM737 [cp737, ibm-737, 737, ibm737]
x-IBM874 [cp874, ibm874, 874, ibm-874]
x-UTF-16LE-BOM [UnicodeLittle]
X-UTF-32BE-BOM [UTF_32BE_BOM, UTF-32BE-BOM]
X-UTF-32LE-BOM [UTF_32LE_BOM, UTF-32LE-BOM]
```