Overture Technical Report Series
No. TR-007
July 2017

VDMJ Tool Support: User Guide

by
Nick Battle
Fujitsu Services
Lovelace Road, Bracknell,
Berkshire. RG12 8SN, 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

0.3. Document History

Issue 0.1	10/10/08	First release.
Issue 0.2	17/10/08	Added threaded debugging.
Issue 0.3	07/11/08	Added the create command, and comments from PGL.
Issue 0.4	26/11/08	Added I18N section, Appendix B and overture option.
Issue 0.5	27/02/09	Added PO generation section, and some new commands.
Issue 0.6	21/04/09	Added the Combinatorial Testing section, and -o libraries
Issue 0.7	04/06/09	Added explanation of hit conditional breakpoints.
Issue 0.8	17/09/09	Added VDM-RT changes and the VDMJC description.
Issue 0.9	14/10/09	Minor changes.
Issue 0.10	30/10/09	Minor changes.
Issue 0.11	09/12/09	Added GPL copyright section.
Issue 1.0	15/01/13	Update for Overture 1.0.0 release and misc corrections
Issue 1.1	18/02/13	Added VDMJUnit section.
Issue 1.2	30/12/15	Updated for VDMJ 3.1.1
Issue 1.3	03/02/16	Added command line recall section
Issue 1.4	08/04/16	Added runtrace ranges
Issue 1.5	29/06/16	Added the runalltraces command
Issue 2.0	26/07/17	Updated for VDMJ version 4

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].

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. Starting VDMJ

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-4.1.0.jar
VDMJ: You must specify either -vdmsl, -vdmpp or -vdmrt

Usage: VDMJ <-vdmsl | -vdmpp | -vdmrt> [<options>] [<files or dirs>]
-vdmsl: parse files as VDM-SL
-vdmpp: parse files as VDM++
-vdmrt: parse files as VDM-RT
-path: search path for files
-v: show VDMJ jar version
-r <release>: VDM language release
-w: suppress warning messages
-q: suppress information messages
-i: run the interpreter if successfully type checked
-p: generate proof obligations and stop
-e <exp>: evaluate <exp> and stop
-c <charset>: select a file charset
-t <charset>: select a console charset
-o <filename>: save the type checked specification
-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
-measures: disable recursive measure checking
-log <filename>: enable real-time event logging
-remote <class>: enable remote control
-verbose: display detailed startup information
```

Notice that the error indicates that the tool must be invoked with either the *-vdmsl, -vdmpp* or *-vdmrt* option to indicate the VDM dialect and parser required. 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 2.2).

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 4).

The *-o* option allows a parsed and type checked specification to be saved to a file. Such files are

effectively libraries, and can be can be re-loaded without the parsing/checking overhead.

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 *-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.

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**:

```
$ 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-4.1.0.jar -v
VDMJ version = 4.1.0 build 170724
```

```
$  java -jar vdmj-4.1.0.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-4.1.0.jar -vdmsl -q -w flat.def
<quiet!>
```

```
$ java -jar vdmj-4.1.0.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-4.1.0.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-4.1.0.jar -vdmsl -e "f(10)" -q -w flat.def
3628800


$ java -jar vdmj-4.1.0.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-4.1.0.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 1 proof obligation:

Proof Obligation 1: (Unproved)
f: function apply obligation in 'DEFAULT' (flat.def) at line 4:38
(forall a:int & (a > 0) =>
  (not (a < 2) =>
    pre_f((a - 1))))


$ java -jar vdmj-4.1.0.jar -vdmsl -e "f(0)" -w flat.def
Parsed 1 module in 0.181 secs. No syntax errors
Type checked 1 module in 0.027 secs. No type errors and suppressed 1
warning
Initialized 1 module in 0.086 secs.
Error 4055: Precondition failure: pre_f in 'DEFAULT' (console) at line 1:1
Execution: Error 4055: Precondition failure: pre_f in 'DEFAULT' (console)
at line 1:1
        a = 0
In root context of f(a) in 'DEFAULT' (console) at line 1:1
In root context of module scope
Bye


$ java -jar vdmj-4.1.0.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

$ java -jar vdmj-4.1.0.jar -vdmsl -w -o flat.lib flat.def
Parsed 1 module in 0.192 secs. No syntax errors
Type checked 1 module in 0.02 secs. No type errors and suppressed 1 warning
Saved 1 module to flat.lib in 0.108 secs.
$ java -jar vdmj-4.1.0.jar -vdmsl flat.lib -e "f(10)"
Loaded 1 module from flat.lib in 0.192 secs
Initialized 1 module in 0.129 secs.
3628800
Bye
```

2.2. 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-4.1.0.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.

2.3. 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[1]. 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.

  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

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

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.

Apart from plain text and LaTeX files (which are the default), VDMJ will recognise the various supported file formats above from the source file name extension: .doc, .docx or .odt.

It is occasionally useful to conditionally include source, depending on the dialect of VDM that is being parsed. 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 defined are VDM_SL, VDM_PP or VDM_RT, depending on the dialect of the parser. For example:

```
#ifdef VDM_SL
values
      val = 123;
#ifdef VDM_PP
      pp = 123;
#else
      notpp = 321;
#endif
#else
instance variables
#ifdef VDM_PP
      pp:int := 123;
#else
      notpp:int := 321;
#endif
#endif
```

As with LaTeX files, lines that are excluded with #ifdefs are present as blank lines in the parsed specification, which enables line numbering to be accurate. It is **not** possible to use #ifdefs with specifications that are embedded in other file formats.

2.4. Parsing, Type Checking, and Proof Obligations

All specification files loaded by VDMJ are parsed and type checked automatically. There are no type checking 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 zero 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 zero.

2.5. 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-4.1.0.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
modules - list the loaded module names
default <module> - set the default module name
state - show the default module state
print <expression> - evaluate expression
runtrace <name> [start test [end test]] - run CT trace(s)
debugtrace <name> [start test [end test]] - debug CT trace(s)
savetrace [<file> | off] - save CT trace output
seedtrace <number> - seed CT trace random generator
runalltraces [<name>] - run all CT traces in class/module name
assert <file> - run assertions from a file
init - re-initialize the global environment
env - list the global symbols in the default environment
pog [<function/operation>] - generate proof obligations
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
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
files - list files in the current specification
set [<pre|post|inv|dtc|measures> <on|off>] - set runtime checks
reload - reload the current specification files
load <files or dirs> - replace current loaded specification files
save [<file>] - generate Word/ODF source extract files
quit - leave the interpreter

> 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 56 proof obligations:

Proof Obligation 1: (Unproved)
sizeof: subtype obligation in 'M' (shmem.vdm) at line 38:1
(forall m:M &
  (((m.stop) - (m.start)) + 1) > 0)
```

```
Proof Obligation 2: (Unproved)
spacefor: recursive function obligation in 'M' (shmem.vdm) at line 48:1
...

Proof Obligation 54: (Unproved)
main: state invariant obligation in 'M' (shmem.vdm) at line 239:11
-- After result := (result ^ [<BEST>])
let mk_Memory(-, q3, q4) = Memory in (((len q3) > 0) and ((len q4) > 0))

Proof Obligation 55: (Unproved)
main: state invariant obligation in 'M' (shmem.vdm) at line 240:11
-- After result := (result ^ [<FIRST>])
let mk_Memory(-, q3, q4) = Memory in (((len q3) > 0) and ((len q4) > 0))
>
```

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-4.1.0.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 *files* command will list the file names that comprise the current loaded specification. The *reload* command will re-parse and check the specification files currently loaded. The *load* command will replace the currently loaded specification with a new set of files. Note that if there are any errors in the parse or type check of the files, the interpreter will exit after *reload* or *load.*

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 and -measures 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
>
```

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

```
$ java -jar vdmj-4.1.0.jar -vdmpp -i
Initialized 1 class in 0.183 secs.
Interpreter started
> help
classes - list the loaded class names
default <class> - set the default class name
```



```
create <id> := <exp> - create a named variable
print <expression> - evaluate expression
runtrace <name> [start test [end test]] - run CT trace(s)
debugtrace <name> [start test [end test]] - debug CT trace(s)
savetrace [<file> | off] - save CT trace output
seedtrace <number> - seed CT trace random generator
runalltraces [<name>] - run all CT traces in class/module name
assert <file> - run assertions from a file
init - re-initialize the global environment
env - list the global symbols in the default environment
pog [<function/operation>] - generate proof obligations
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
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
files - list files in the current specification
set [<pre|post|inv|dtc|measures> <on|off>] - set runtime checks
reload - reload the current specification files
load <files or dirs> - replace current loaded specification files
save [<files>] - generate Word/ODF source extract files
quit - leave the interpreter
>
```

The *classes* command replaces the *modules* command, but is similar in that it lists the names of the classes loaded, identifying the default. 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-4.1.0.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 not available in this context
> 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
a = A{#4, si:=undefined, sj:=undefined, sk:=579}
> p a.op(111,222)
= 333
Executed in 0.0020 secs.
> env
test() = (() ==> int)
sk = 333
a = A{#4, si:=111, sj:=222, sk:=333}
> quit
Bye
```

Executing a specification written in the VDM-RT dialect is essentially the same as with VDM++. There is one additional command, *log*, which is used to indicate where real-time events from the interpreter should be sent:

```
> 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
...
```

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
```

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>.*

2.5.1. 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* and *-measures* options to selectively suppress 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.

2.6. Debugging

2.6.1. 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 2.4, we can stop on the *op* operation as follows:

```
> break op
Created break [1] in 'A' (test.vpp) at line 12:3
12:     ( si := i;
> p test()
Stopped break [1] in 'A' (test.vpp) at line 12:3
12:     ( si := i;
[MainThread-7]> help
step - step one expression/statement
next - step over functions or operations
out - run to the return of functions or operations
...
load <files> - replace current loaded specification files
quit - leave the interpreter
[MainThread-7]> stack
Stopped at break [1] in 'A' (test.vpp) at line 12:3
        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
[MainThread-7]> 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-7]> down
In context of let statement in 'A' (test.vpp) at line 19:15
[MainThread-7]> stack
Stopped at break [1] in 'A' (test.vpp) at line 12:3
        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
[MainThread-7]> print a
a = A{#1, si:=undefined, sj:=undefined, sk:=123}
[MainThread-7]> step
Stopped in 'A' (test.vpp) at line 13:5
13:      sj := j;
```

```
[MainThread-7]> step
Stopped in 'A' (test.vpp) at line 14:5
14:       sk := i + j;
[MainThread-7]> step
Stopped in 'A' (test.vpp) at line 15:5
15:       return sk;
[MainThread-7]> step
= 3
Executed in 83.143 secs.
> list
break [1] in 'A' (test.vpp) at line 12:3
12:     ( si := i;
> break op i < 10
Created break [2] when "i < 10" in 'A' (test.vpp) at line 12:3
12:     ( si := i;
> print new A().op(1,2)
Stopped break [2] when "i < 10" in 'A' (test.vpp) at line 12:3
12:     ( si := i;
[MainThread-8]> 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 12 = 4
Created break [1] when "= 4" in 'A' (test.vpp) at line 12:3
12:     ( si := i;
> break 12 > 4
Created break [2] when "> 4" in 'A' (test.vpp) at line 12:3
12:     ( si := i;
> break 12 >= 4
Created break [3] when ">= 4" in 'A' (test.vpp) at line 12:3
12:     ( si := i;
> break 12 mod 4
Created break [4] when "mod 4" in 'A' (test.vpp) at line 12:3
12:     ( si := i;
> break 12 i > j or i+j > 10
Created break [5] when "i > j or i+j > 10" in 'A' (test.vpp) at line 12:3
12:     ( 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-7]> 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-7]> p x
x = -1
[MainThread-7]> p y
y = 0
[MainThread-7]> continue
Runtime: Error 4134: Infinite or NaN trouble in 'A' (coverage.vpp) at line
5:11
>
```

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.

2.6.2. VDM++ Multi-threaded Debugging

The examples in section 2.5.1 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 is an extra command available, “threads”, which displays the status of all active threads.

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 the current thread and allow another thread to be controlled from the console. There is a race to gain control of the console at a breakpoint. When this happens, 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 which print out their thread IDs every 1000 ticks (see the listing at the end of the section):

```
> break op
Created break [1] in 'A' (threads.vpp) at line 5:9
5:          (
> p new A().run()
Stopped in 'A' (threads.vpp) at line 14:27
14:               while true do skip
[MainThread-7]> threads
PeriodicThread-9 (RUNNABLE)
PeriodicThread-10 (RUNNABLE)
MainThread-7 (RUNNABLE)
PeriodicThread-8 (RUNNING)
PeriodicThread-11 (RUNNABLE)
[MainThread-7]> c
Stopped break [1] in 'A' (threads.vpp) at line 5:9
5:          (
[PeriodicThread-8]> c
Thread 8...
Stopped in 'A' (threads.vpp) at line 14:27
14:               while true do skip
[MainThread-7]> c
Stopped break [1] in 'A' (threads.vpp) at line 5:9
5:          (
[PeriodicThread-9]> c
Thread 9...
Stopped in 'A' (threads.vpp) at line 14:27
14:               while true do skip
[MainThread-7]> c
Stopped break [1] in 'A' (threads.vpp) at line 5:9
5:          (
[PeriodicThread-10]> c
Thread 10...
Stopped in 'A' (threads.vpp) at line 14:27
14:               while true do skip
[MainThread-7]> c
Stopped break [1] in 'A' (threads.vpp) at line 5:9
5:          (
```

```
[PeriodicThread-11]> c
Thread 11...
Stopped in 'A' (threads.vpp) at line 14:27
14:               while true do skip
[MainThread-7]> c
Stopped break [1] in 'A' (threads.vpp) at line 5:9
5:          (
[PeriodicThread-12]> c
Thread 12...
Stopped in 'A' (threads.vpp) at line 14:27
14:               while true do skip
[MainThread-7]> remove 1
Cleared break [1] in 'A' (threads.vpp) at line 5:9
5:          (
[MainThread-7]> c
Stopped break [1] in 'A' (threads.vpp) at line 5:9
5:          (
[PeriodicThread-13]> c
Thread 13...
Thread 14...
Thread 15...
Thread 16...
Thread 17...
```

The operation, “op”, calls the IO class to print out its thread ID, followed by a skip statement. This is then executed periodically by three periodic threads (see specification below). The periodic threads each call op and create another periodic thread to run 1000 ticks later.

The breakpoint on “op” first stops thread 8 as the “threads” command shows, but thread 7 has grabbed the console. Behind the scenes, the periodic threads 9, 10 and 11 are suspended – they just don’t have control of the console. So a series of “continue” commands cycles through the threads that are have been suspended, and follows the resource scheduler as it swaps the various threads. The “threads” command reveals the state of the threads at any time.

After the breakpoint is removed, one more continue catches thread 13 (which was trying to acquire the console), and thereafter the periodics act as expected, printing out successive thread IDs.

This bizarre looking control sequence is necessary to avoid a debugging session producing a different outcome (by a different thread ordering) to a real execution. The threaded resource scheduling is deterministic in VDMJ, and that determinism is preserved in the debugger.

```
operations
      public op: () ==> ()
            op() ==
            (
                  IO`printf("Thread %s...", [threadid]);
                  skip
            );

      public run: () ==> ()
            run() ==
            (
                  startlist({new A(), new A(), new A()});
                  while true do skip
            );

thread
      periodic (1000)(op)
```


2.6.3. VDM-RT Debugging and the VDMJC Client

The VDM-RT dialect is specialized for the specification of distributed multi-processor real-time systems. This means that, while the majority of the language is the same as VDM++, there are special constructs which allow a VDM-RT specification to define the CPU and BUS topology of the system, as well as the deployment of VDM objects to CPUs.

Debugging VDM-RT specifications is therefore almost always about multi-threaded debugging, and the weaknesses of the console-based multi-threaded debugging are even more problematic than they are for threaded VDM++ (see 2.4.2).

To counter these problems, and to provide better multi-threaded debugging for GUI based systems, VDMJ includes an alternative method to control the execution and debugging of a specification. This uses a remote control protocol (the client and the interpreter are in different processes). A separate client called *VDMJC* (VDMJ Client) is provided to act as an initiator, to pass user commands to the interpreter using the remote protocol and to display results from multiple threads.

Running the VDMJC client is very easy compared to all the confusing options for VDMJ itself

```
$ java -jar vdmjc-4.1.0.jar <-vdmsl | -vdmpp | -vdmrt> [command]
```

The behaviour can be changed by settings in a *vdmjc.properties* file (which must be on the classpath), rather than by passing lots of command line options. But by default, the client looks for the VDMJ jar file in the current directory.

```
# Properties for the VDMJ client

vdmj.jar = ../core/vdmj/target/vdmj-4.1.0.jar
vdmj.jvm = -Xmx1024m
```

You can use VDMJC for any VDM dialect, but it is most effective for VDM++ and VDM-RT with threads. After starting the client, the dialect is displayed for information, but no VDMJ interpreter is running. The *help* command will always show you what you can do:

```
Dialect is VDM_RT
> help
Loading and starting:
  load [<files>]
  eval [<files>]
  dbgp
  quiet
  help
  ls | dir
  q[uit]

Use 'help <command>' for more help
>
```

To load a specification, starting an instance of VDMJ, you use the *load* command:

```
> load vice.vpp
Parsed 2 classes in 0.312 secs. No syntax errors
Warning 5000: Definition 'i' not used in 'A' (vice.vpp) at line 73:13
Warning 5000: Definition 'i' not used in 'A' (vice.vpp) at line 80:13
Type checked 4 classes in 0.016 secs. No type errors and 2 warnings
[Id ?: STARTING]>
Standard output redirected to client
```

```
Standard error redirected to client
[Id 1: STARTING]>
```

The connection between the client and the VDMJ instance is completely asynchronous, so it is possible for VDMJ to "say something" while you are typing, and for you to type commands while VDMJ is operating. Notice in the example above that there are two prompts. The first is immediately after VDMJ has started, where the client does not know the main thread's ID. Then commands are sent to redirect the input and output back to the VDMJC client, which causes the main thread to identify itself as ID 1, and the prompt is repeated.

Simple evaluations can then be performed in the usual way using the *print* command:

```
[Id 1: STARTING]> p new A().Get()
[Id 1: RUNNING]>
new A().Get() = 100
[Id 1: STOPPED]>
```

Notice that the prompt changed to indicate that VDMJ was running, but then the result was returned and the prompt indicated that the evaluation had stopped. This is another example of the asynchronicity of the client and interpreter. If the evaluation had taken a very long time, the client could examine the threads that were running, and switch between them as individual threads reach a breakpoint and so on. This is much more flexible than the command line debugger described in section 2.5.1.

Note that the final state of the system is STOPPED rather than STARTING now. This reflects the way the remote debugging protocol works[2]. It expects to have a single "main" evaluation that is STARTING at the start of the session, then RUNNING or BREAK at breakpoints, finally reaching the STOPPED state at the end. This does not mean that further evaluations cannot be performed however:

```
[Id 1: STOPPED]> p obj1
[Id 1: RUNNING]>
obj1 = A{#2, val:=0}

[Id 1: STOPPED]> p obj1.Set(123)
[Id 1: RUNNING]>
obj1.Set(123) = ()
[Id 1: STOPPED]>

[Id 1: STOPPED]> p obj1
[Id 1: RUNNING]>
obj1 = A{#2, val:=123}
[Id 1: STOPPED]>
```

This example shows a (static) object's value being printed, followed by a call to Set(123) which is an async VDM-RT operation. This creates a new async thread, though no message is seen on the client unless that new thread stops (eg. at a breakpoint).

```
async public Set: nat ==> ()
Set(n) == val := n;
```

To illustrate threaded debugging, we use the *factorial* example from the CSK VDM++ manual.

```
[Id 1: STARTING]> break 35
Breakpoint [1] set
[Id 1: STARTING]> p new Factorial().factorial(2)
[Id 1: RUNNING]>
```

2 The protocol is the Xdebug protocol, called DBGP. This is a standard protocol and allows VDMJ to be integrated with Eclipse.


```
[Id 1: BREAK]>
New thread: Id 8: STARTING
[Id 1: BREAK]>
Stopped at in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 59:35
59:   per giveResult => #fin (doit) > #act (giveResult);
[Id 1: BREAK]>
Stopped at break [1] in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 35:9
35:         if i = j then result := i
[Id 1: BREAK]> threads
Id 1: BREAK
Id 8: BREAK
[Id 1: BREAK]> thread 8
[Id 8: BREAK]> p i
[Id 8: BREAK]>
i = 1
[Id 8: BREAK]> p j
[Id 8: BREAK]>
j = 2
[Id 8: BREAK]> c
[Id 8: RUNNING]> thread 1
[Id 1: BREAK]> c
[Id 1: RUNNING]>
[Id 1: BREAK]>
New thread: Id 9: STARTING
[Id 1: BREAK]>
Stopped at in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 59:35
59:   per giveResult => #fin (doit) > #act (giveResult);
Stopped at in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 59:35
59:   per giveResult => #fin (doit) > #act (giveResult);
[Id 1: BREAK]>
Stopped at break [1] in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 35:9
35:         if i = j then result := i
[Id 1: BREAK]> threads
Id 1: BREAK
Id 8: BREAK
Id 9: BREAK
[Id 1: BREAK]> thread 9
[Id 9: BREAK]> p j
[Id 9: BREAK]>
j = 1
[Id 9: BREAK]> c
[Id 9: RUNNING]> thread 8
[Id 8: BREAK]> c
[Id 8: RUNNING]>
Thread stopped: Id 9: STOPPED
[Id 8: RUNNING]> thread 1
[Id 1: BREAK]> c
[Id 1: RUNNING]>
[Id 1: BREAK]>
New thread: Id 10: STARTING
[Id 1: BREAK]>
Stopped at in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 59:35
59:   per giveResult => #fin (doit) > #act (giveResult);
Stopped at in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 59:35
59:   per giveResult => #fin (doit) > #act (giveResult);
[Id 1: BREAK]>
Stopped at break [1] in 'Multiplier'
(/home/nick/eclipse.overture/VDMJTests/factorial.vpp) at line 35:9
```

```
35:         if i = j then result := i
[Id 1: BREAK]> threads
Id 1: BREAK
Id 8: BREAK
Id 10: BREAK
[Id 1: BREAK]> thread 10
[Id 10: BREAK]> c
[Id 10: RUNNING]> thread 8
[Id 8: BREAK]> c
[Id 8: RUNNING]>
Thread stopped: Id 10: STOPPED
[Id 8: RUNNING]> thread 1
[Id 1: BREAK]> c
[Id 1: RUNNING]>
Thread stopped: Id 8: STOPPED
new Factorial().factorial(2) = 2
[Id 1: RUNNING]>
[Id 1: STOPPED]>
```

Notice that now, the threads' creation and destruction are visible in the client because the threads are stopping and may need to interact with the user.

Line 35 is the important "decision point" in the algorithm where the recursion either continues and forks another pair of Multiplier threads, or returns the result from this level. The first breakpoint is when there are only two threads: the main thread 1, and the first Multiplier thread, 8. The *threads* command shows that both threads have stopped because of the breakpoint, and the *thread 8* command switches the debugger's context to the thread that has stopped. From there, the stack and the local variables can be printed, expressions evaluated etc. Each thread must then be continued by hand. The evaluation stops twice more at the same breakpoint, and each time we switch to the thread concerned, print out a value and continue all threads. Eventually the final result is printed, as expected.

This is confusing, as with the simple command line debugger. But with VDMJC, as with a GUI, you can switch to any thread context you wish at any time.

Similarly, tracepoints can be used to display variables without stopping:

```
[Id 1: STOPPED]> trace 35 mk_(i,j)
[Id 1: STOPPED]>
Created trace [2] show "mk_(i,j)" in 'Multiplier' at line 35:9
35:         if i = j then result := i
[Id 1: STOPPED]> p new Factorial().factorial(3)
[Id 1: RUNNING]>
[Id 11: RUNNING] mk_(i,j) = mk_(1, 3) at [2]
[Id 1: RUNNING]>
[Id 12: RUNNING] mk_(i,j) = mk_(1, 2) at [2]
[Id 13: RUNNING] mk_(i,j) = mk_(3, 3) at [2]
[Id 14: RUNNING] mk_(i,j) = mk_(1, 1) at [2]
new Factorial().factorial(3) = 6
[Id 1: STOPPED]>
[Id 15: RUNNING] mk_(i,j) = mk_(2, 2) at [2]
[Id 1: STOPPED]>
```

Notice that here again, the asynchronous nature of the link to VDMJ means that the final mk_(2,2) output happens to arrive after the print of the final result. This is not because any of the factorial operations are asynchronous, but because by chance, the interleaving of the output from the multiple threads arrived that way – the VDMJC client will not interleave output on a single line, so messages from separate threads are always distinct.

To finish a VDMJ session, use the *quit* command:

```
[Id 1: STOPPED]> quit
Terminating VDMJ process
```



```
>
```

This returns to the original VDMJC prompt, as no specification is now loaded. It does not leave the client immediately (a further *quit* will do that).

The other way to start a specification from this point is to use the *eval* command rather than *load*. This is very similar, except an expression is prompted for at the start, and it is executed via the *run* command, the result appearing on standard output. There is no real advantage to using this from the command line, but it is useful for GUI debuggers which want to create a session related to the evaluation of a single specific expression, and get the result.

```
> eval factorial.vpp
Evaluate: new Factorial().factorial(4)
Parsed 2 classes in 0.328 secs. No syntax errors
Type checked 4 classes in 0.015 secs. No type errors and 4 warnings
[Id 1: STARTING]>
Standard output redirected to client
Standard error redirected to client
[Id 1: STARTING]> run
[Id 1: RUNNING]>
stdout: 24
[Id 1: STOPPED]>
```

2.7. 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.
> threads
ObjectThread-10 (WAITING)
> p a.get()
= 1
Executed in 0.019 secs.
> threads
ObjectThread-10 (RUNNABLE)
> p a
= A{#1, iv:=2}
Executed in 0.0080 secs.
> threads
ObjectThread-10 (WAITING)
```

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()*.

Note that immediately after a call to *get*, the thread is in state RUNNABLE. This is because as soon as the *get* operation has been called, its *#fin* counter is incremented and the thread is then immediately RUNNABLE as it now passes the permission predicate. However, execution returns to the command line immediately after the *get* operation so the thread scheduler will not actually run the thread (ie. call *inc*) until the next *get(a)* call (or any other operation that resumes the model). If we resume the model with a simple *print a* command, the background thread resumes but finds that after the increment it must wait for the next *get* call, and so enters state WAITING as expected.

2.8. 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.

To use a library, your specification must include one or more of the VDM source files for the library. These provide a bridge from VDM to the native Java library, using the facilities for making native calls built into VDMJ (see [10]). The VDM library source files were originally written by CSK Systems Inc. as part of *VDMTools*, but they are distributed freely with the Overture project.

For example, the following simple VDM function calls the sine function from the MATH library, assuming the VDM-SL source is located in *stdlib/MATH.vdm*. Note that the test uses the constant *MATH`pi*.

```
functions
      sin: real -> real
      sin(radians) == MATH`sin(radians);
```

```
$ java -jar vdmj-4.1.0.jar -vdmsl -i test.vdm stdlib/MATH.vdm
Parsed 2 modules in 0.282 secs. No syntax errors
Type checked 2 modules in 0.016 secs. No type errors
Initialized 2 modules in 0.062 secs.
Interpreter started
> p sin(MATH`pi/4)
= 0.7071067811865475
Executed in 0.016 secs.
>
```

Note that the *-path* command line option can be useful to specify the location of a common library folder that includes the standard library VDM sources. Library sources can then be referred to without the path prefix.

3. VDMJUnit Testing

Section 2.6 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.

```
@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:

```
@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:

```
@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.
- **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.

```
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

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-396]> p total
total = 0
[CTMainThread-396]> c
Stopped break [1] in 'Tested' (test.vpp) at line 10:18
10:         op(a) == ( total := total + a; return total; );
[CTMainThread-396]> p total
total = 222
[CTMainThread-396]> p a
a = 222
[CTMainThread-396]> 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-400]> 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-11]> p a
a = 1
[CTMainThread-11]> p b
b = 0
[CTMainThread-11]> stack
Stopped [CTMainThread-11] 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-11]> 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. 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-4.1.0.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-4.1.0.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-4.1.0.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-4.1.0.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.

6. The vdmj.properties File

The command line options described in section 2.1 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 default vdmj.properties file is as follows:

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

# The tab stop for source files.
# (default 4)
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.

```
# The maximum number of expansions for "+" and "*" trace patterns.
# (default 5)
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.

```
# The default duration for RT statements
# (default 2)
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".

```
# The default timeslice (statements executed) for the FCFS policy
# (default 10)
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)
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)
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.

```
# Enable transactional variable updates
# (default false)
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)
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)
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)
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.

7. Appendix A: The shmem Example

The following is the source of the *shmem.vdm* example used in section 2.4. 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
```

8. 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]
```