VDMJ User Guide

Author	Nick Battle
Date	05/03/09
Issue	0.6

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.

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	05/03/09	Added the Combinatorial Testing section.

1. Introduction

VDMJ provides computer based tool support for the VDM-SL and VDM++ specification languages, written in Java. The tool includes a parser, a type checker, an interpreter, a proof obligation generator and a debugger. It is a command line tool only.

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_0.1_090114.jar
VDMJ: You must specify either -vdmsl, -vdmpp or -overture

Usage: VDMJ <-vdmsl | -vdmpp | -overture> [<options>] [<files>]
-vdmsl: parse files as VDM-SL
-vdmpp: parse files as VDM++
-overture: parse files as VDM++ with Overture
-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
-pre: disable precondition checks
-post: disable postcondition checks
-inv: disable type/state invariant checks
-dtc: disable all dynamic type checking
```

Notice that the error indicates that the tool must be invoked with either the *-vdmsl, -vdmpp* or *-overture* option to indicate the VDM dialect and parser required. Using the Overture parser should produce identical results to *-vdmpp*, though it will be slower.

Normally, a specification will be loaded by identifying all of the VDM source files to include. 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.

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 a specification written in languages other than the default for your system to be used (see section 3).

The *-pre, -post, -inv* and *-dtc* options can be used to disable precondition, postcondition, invariant and dynamic type checking, respectively. By default, all these checks are performed.

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_0.1_090114.jar -vdmsl flat.def
Parsed 1 module in 0.202 secs. No syntax errors
Warning 5012: Recursive function has no measure in (flat.def) at line 3:1
Type checked 1 module in 0.016 secs. No type errors and 1 warning
```

```
$ java -jar vdmj_0.1_090114.jar -vdmsl -q -w flat.def
<quiet!>
```

```
$ java -jar vdmj_0.1_090114.jar -vdmsl -w -e "f(10)" flat.def
Parsed 1 module in 0.28 secs. No syntax errors
Type checked 1 module in 0.031 secs. No type errors, suppressed 1 warning
Initialized 1 module in 0.031 secs.
3628800
Bye
```

```
$ java -jar vdmj_0.1_090114.jar -vdmsl -e "f(10)" -q -w flat.def
3628800
```

```
$ java -jar vdmj_0.1_090114.jar -vdmsl -i -w flat.def
Parsed 1 module in 0.202 secs. No syntax errors
Type checked 1 module in 0.016 secs. No type errors, suppressed 1 warning
Initialized 1 module in 0.031 secs.
Interpreter started
> print f(10)
= 3628800
Executed in 0.0 secs.
> quit
Bye
```

```
$ java -jar vdmj_0.1_090114.jar -vdmsl -p -w flat.def
Parsed 1 module in 0.218 secs. No syntax errors
Type checked 1 module in 0.015 secs. No type errors, suppressed 1 warning
Generated 1 proof obligation:

Proof Obligation 1:
f: function apply obligation in 'DEFAULT1' (flat.def) at line 4:38
(forall a:int & (a > 0) =>
  (not (a < 2) =>
    pre_f((a - 1))))
```

```
$ java -jar vdmj_0.1_090114.jar -vdmsl -w -e "f(0)" -w flat.def
Parsed 1 module in 0.203 secs. No syntax errors
Type checked 1 module in 0.015 secs. No type errors, suppressed 1 warning
Initialized 1 module in 0.016 secs.
Execution: Error 4055: Precondition failure: pre_f in (flat.def) at line
5:11
```



```
        a = 0
        f = (int -> int)
        pre_f = (int +> bool)
In root context of f(a) in 'DEFAULT1' (console) at line 1:1
In root context of interpreter in 'DEFAULT1' (flat.def) at line 3:1
In root context of global environment
Bye
```

```
$ java -jar vdmj_0.1_090114.jar -vdmsl -w -pre -e "f(0)" -w flat.def
Parsed 1 module in 0.218 secs. No syntax errors
Type checked 1 module in 0.016 secs. No type errors, suppressed 1 warning
Initialized 1 module in 0.015 secs.
1
Bye
$
```

2.2. 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.

A source file may contain VDM embedded in a LaTeX file; the markup is ignored by the parser, though reported line numbers will be correct.

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.

2.3. 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_0.1_090114.jar -vdmsl -i shmem.vdm
Parsed 1 module in 0.266 secs. No syntax errors
Type checked in 0.047 secs. No type errors
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
assert <file> - run assertions from a file
init - re-initialize the global environment
env - list the global symbols in the default environment
pog - generate a list of 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 [<file>|clear] - display/clear file line coverage
files - list files in the current specification
reload - reload the current specification files
load <files> - replace current loaded specification files
quit - leave the interpreter

> modules
M (default)

> state
M`Q4 = [mk_M(<FREE>, 0, 9999)]
M`rseed = 87654321
M`Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)], [mk_M(<FREE>, 0,
9999)])
M`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
M`Q4 = [mk_M(<FREE>, 0, 9999)]
M`rseed = 566044643
M`Memory = mk_Memory(566044643, [mk_M(<FREE>, 0, 9999)], [mk_M(<FREE>, 0,
9999)])
M`Q3 = [mk_M(<FREE>, 0, 9999)]

> init
Global context initialized

> state
M`Q4 = [mk_M(<FREE>, 0, 9999)]
M`rseed = 87654321
M`Memory = mk_Memory(87654321, [mk_M(<FREE>, 0, 9999)], [mk_M(<FREE>, 0,
9999)])
M`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
M`fragments = (M`Quadrant -> nat)
M`combine = (M`Quadrant -> M`Quadrant)
M`tryBest = (nat ==> nat)
M`seed = (nat1 ==> ())
M`reset = (() ==> ())
M`bestfit = (nat1 * M`Quadrant -> nat1)
M`add = (nat1 * nat1 * M`Quadrant -> M`Quadrant)
M`firstFit = (nat1 ==> bool)
M`rand = (nat1 ==> nat1)
M`tryFirst = (nat ==> nat)
M`main = (nat1 * nat1 ==> seq of (<SAME> | <BEST> | <FIRST>))
M`MAXMEM = 10000
M`delete = (M`M * M`Quadrant -> M`Quadrant)
M`inv_M = (M`M +> bool)
```



```
M`CHUNK = 100
M`bestFit = (nat1 ==> bool)
M`least = (nat1 * nat1 -> nat1)
M`fits = (nat1 * M`Quadrant -> nat1)
M`init_Memory = (M`Memory +> bool)
M`pre_add = (nat1 * nat1 * M`Quadrant +> bool)

> pog
Generated 36 proof obligations:

Proof Obligation 1:
M`fits: cases exhaustive obligation in 'M' (shmem.vdm) at line 40:5
(forall size:nat1, Q:Quadrant &
  Q = [] or Q = [h] ^ tail)

...

Proof Obligation 35:
M`tryBest: sequence apply obligation in 'M' (shmem.vdm) at line 176:27
rand((len Q4)) in set inds Q4

Proof Obligation 36:
M`tryBest: subtype obligation in 'M' (shmem.vdm) at line 166:1
RESULT >= 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.

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_0.1_090114.jar -vdmsl -i shmem.vdm
Parsed 1 module in 0.297 secs. No syntax errors
Type checked in 0.031 secs. No type errors
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
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
>
>
```

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

```
$ java -jar vdmj_0.1_090114.jar -vdmpp -i
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
assert <file> - run assertions from a file
init - re-initialize the global environment
env - list the global symbols in the default environment
pog - generate a list of 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 [<file>|clear] - display/clear file line coverage
files - list files in the current specification
reload - reload the current specification files
load <files> - replace current loaded specification 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_0.1_090114.jar -vdmpp -i test.vpp
Parsed 1 class in 0.281 secs. No syntax errors
Warning 5001: variable is not initialized: si in 'A' (test.vpp) at line 4:3
Warning 5001: variable is not initialized: sj in 'A' (test.vpp) at line 5:3
Type checked 1 class in 0.016 secs. No type errors and 2 warnings
Initialized 1 class in 0.015 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 0.016 secs.
> env
test() = (() ==> int)
sk = 123
> print test()
= 3
Executed in 0.0 secs.
> env
test() = (() ==> int)
sk = 3
> p new A().op(123,456)
= 579
```



```
Executed in 0.0 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 0.0 secs.
> env
test() = (() ==> int)
sk = 333
a = A{#4, si:=111, sj:=222, sk:=333}
> quit
Bye
```

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 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 = 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.

Coverage information is reset when a specification is loaded, 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*>.

Note that coverage of source files that contain LaTeX markup is printed *without* the markup.

2.3.1. Suppressing Runtime Checks

By default, the interpreter will evaluate all preconditions, postconditions, state and type invariants, and perform dynamic type checking when values are manipulated in a specification. 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* and *-dtc* options to selectively suppress them.

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++). 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).

2.4. Debugging

2.4.1. 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 above, 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;
[thread 1]> help
step - step one expression/statement
next - step over functions or operations
out - run to the return of functions or operations
continue - resume execution
stack - display the current stack frame context
up - move the stack frame context up one frame
down - move the stack frame context down one frame
source - list VDM source code around the current breakpoint
stop - terminate the execution immediately
...

[thread 1]> stack
Stopped at break [1] in 'A' (test.vpp) at line 12:3
      j = 2
      i = 1
      A`self = A{#1, si:=undefined, sj:=undefined, sk:=123}
In object context of op(i, j) in 'A' (test.vpp) at line 10:8
In root context of test() in 'A' (test.vpp) at line 18:15
In root context of global static scope
[thread 1]> 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:
[thread 1]> down
In context of let statement in 'A' (test.vpp) at line 19:15
[thread 1]> stack
Stopped at break [1] in 'A' (test.vpp) at line 12:3
      a = A{#2, si:=undefined, sj:=undefined, sk:=123}
In context of let statement in 'A' (test.vpp) at line 19:15
In root context of test() in 'A' (test.vpp) at line 18:15
```



```
In root context of global static scope
[thread 1]> print a
a = A{#2, si:=undefined, sj:=undefined, sk:=123}
[thread 1]> step
Stopped in 'A' (test.vpp) at line 13:5
13:      sj := j;
[thread 1]> step
Stopped in 'A' (test.vpp) at line 14:5
14:      sk := i + j;
[thread 1]> step
Stopped in 'A' (test.vpp) at line 15:5
15:      return sk;
[thread 1]> step
= 3
Executed in 39.122 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;
[thread 1]> continue
= 3
Executed in 3.5 secs.
> print new A().op(100,200)
= 300
Executed in 0.0 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.

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:

```
> list
break [2] when "i < 10" in 'A' (test.vpp) at line 12:3
12:    ( si := i;
> trace 14 sk
Created trace [3] show "sk" in 'A' (test.vpp) at line 14:5
14:      sk := i + j;
> remove 2
Cleared break [2] when "i < 10" in 'A' (test.vpp) at line 12:3
12:    ( si := i;
> p test()
sk = 300 at [3]
= 3
Executed in 0.0 secs.
>
```

Note that new breakpoints and tracepoints can be set when stopped in the debugger (ie. when the command prompt is “[thread 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(1,2)
= 0.5
Executed in 0.0 secs.
> p new A().f(2,0)
Runtime: Error 4134: Infinite trouble in 'A' (test.vpp) at line 4:17
Stopped in 'A' (test.vpp) at line 4:17
4:    f(x, y) == x/y
[thread 1]> source
1:  class A
2:  functions
3:  public f: int * int -> real
4:>>  f(x, y) == x/y
5:
6:  end A
7:  ~
[thread 1]> p x
x = 2
[thread 1]> p y
y = 0
[thread 1]> continue
>
```

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.

2.4.2. Multi-threaded Debugging

The examples in section 2.4.1 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 "[Thread 1]".

VDM++ specifications can create multiple threads, and the debugger provides some support for multi-threaded debugging.

The basic principles and most debugging commands are the same as the single threaded case. There is an extra command available, “threads”, which displays the status of any active threads (ie. those in addition to the main thread, which is always number 1).

When a breakpoint is reached, all threads will be suspended. Single stepping will then advance the current thread, while the others remain blocked. Using the “continue” command will resume all threads. In practice, there is a race to gain control of the console at a breakpoint, and multiple threads can reach the same breakpoint together. When this happens, single stepping may flip to one of the other threads, which can be confusing. The current thread is 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 techniques – the specification contains three threads which print out their thread IDs once a second (see the listing at the end of the section):


```
> break op
Created break [1] in 'A' (test.vpp) at line 35:9
35:         (
> p new A().run()
Stopped break [1] in 'A' (test.vpp) at line 35:9
35:         (
[thread 9]> n
Stopped in 'A' (test.vpp) at line 36:17
36:             new IO().print("Thread " ^ str(threadid) ^ "...");
[thread 9]> n
Stopped break [1] in 'A' (test.vpp) at line 35:9
35:         (
[thread 8]> n
Stopped in 'A' (test.vpp) at line 36:17
36:             new IO().print("Thread " ^ str(threadid) ^ "...");
[thread 8]> n
Stopped break [1] in 'A' (test.vpp) at line 35:9
35:         (
[thread 10]> n
Stopped in 'A' (test.vpp) at line 36:17
36:             new IO().print("Thread " ^ str(threadid) ^ "...");
[thread 10]> n
Thread 10...
Stopped in 'A' (test.vpp) at line 37:13
37:             skip
[thread 10]> n
Stopped break [1] in 'A' (test.vpp) at line 35:9
35:         (
[thread 10]> n
Stopped in 'A' (test.vpp) at line 36:17
36:             new IO().print("Thread " ^ str(threadid) ^ "...");
[thread 10]> n
Thread 10...
Stopped in 'A' (test.vpp) at line 37:13
37:             skip
[thread 10]> threads
Thread 8, self #2, class A, state SUSPENDED
Thread 9, self #3, class A, state SUSPENDED
Thread 10, self #4, class A, state SUSPENDED
[thread 10]> c
Thread 8...
Thread 9...
Stopped break [1] in 'A' (test.vpp) at line 35:9
35:         (
[thread 10]> remove 1
Cleared break [1] in 'A' (test.vpp) at line 35:9
35:         (
[thread 10]> c
Stopped in 'A' (test.vpp) at line 37:13
37:             skip
[thread 9]> c
Stopped in 'A' (test.vpp) at line 37:13
37:             skip
[thread 8]> c
Thread 10...
Thread 9...
Thread 8...
Thread 10...
Thread 9...
Thread 8...
Thread 10...
Thread 9...
Thread 8...
```


...etc

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 threads, numbers 8, 9 and 10 (see specification below).

The breakpoint on “op” first stops thread 9. But behind the scenes, threads 8 and 10 have also stopped at the breakpoint – they just don’t have control of the console. So a series of “next” commands eventually reveals all three threads have stopped at the same place. Thread 10 is the one to carry on after more “next” commands and actually print out “Thread 10...”, and subsequent “next” commands would only advance thread 10 (ie. threads 8 and 9 are now suspended). The “threads” command reveals this.

A “continue” lets threads 8 and 9 print out their messages, and all threads resume before stopping at the breakpoint again, which happens to be in thread 10 first. The breakpoint is then removed, and a “continue” given, but once again threads 8 and 9 have also reached the breakpoint (just removed), and so they gain control of the console immediately. Two further “continue” commands allows all threads to continue, and the expected output scrolls up the screen.

```
operations
      public op: () ==> ()
            op() ==
            (
                  new IO().print("Thread " ^ str(threadid) ^ "...");
                  skip
            );

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

thread
      periodic (1000)(op)
```


3. 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 known as *Combinatorial Testing* [7].

The idea behind combinatorial testing is that classes can be tested by making a sequence of calls on the public interface of an instance of the class, 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 may defined 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. In VDMJ, each of the two traces (test1 and test2) causes a public static operation to be defined in the class with the same name; calling these operations causes the trace definitions to be executed.

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

```
> p Tester`test1()
Test 1 = obj.op1()
Result = [1, PASSED]
= ()
Executed in 0.016 secs.

> p Tester`test2()
Test 1 = obj.op1(); obj.op1(); obj.op1()
Result = [1, 2, 3, PASSED]
= ()
Executed in 0.0 secs.
```

Each operation 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 step, 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:

```
> p Tester`test2()
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 0.016 secs.
```

Notice that now there are five tests listed, Test 1 to Test 5, and that they each have one more op1 call than the previous. 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:

```
> p Tester`test3()
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 0.032 secs.
```

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;
...
```

```
> p Tester`test3()
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 'Tester'
(traces.vpp) at line 29:29, 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 'Tester'
(traces.vpp) at line 29:29, FAILED]
Test 5 = obj.Tested`op1(); obj.Tested`op2(4, 5); obj.Tested`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).

4. 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_0.1_081110.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_0.1_081110.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_0.1_081110.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_0.1_081110.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.

5. Appendix A: The shmem Example

The following is the source of the *shmem.vdm* example used in section 2.3. 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.

Note that it can be very CPU intensive:

```
> p main(5, 100)
= [<FIRST>, <SAME>, <FIRST>, <BEST>, <FIRST>]
Executed in 34.344 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: int
      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;

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

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;

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;

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);

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

		return result;
	)

	end M
```


6. 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]
```

