1. Debugging with the DBGP Client

VDMJ includes an alternative method to control the execution and debugging of a specification. This uses a remote control protocol called DBGP (the client and the interpreter are in different processes). A separate client is provided to act as an initiator, to pass user commands to the interpreter using the DBGP protocol and to display results from multiple threads.

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

```
$ java -jar dbgpc-4.4.0.jar <-vdmsl | -vdmpp | -vdmrt> [command]
```

The behaviour can be changed by settings in a *dbgpc.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 and DBGP protocol jar files in the current directory.

```
# Properties for the DBGP client

dbgpc.vdmj_jar = ../vdmj/target/vdmj-4.4.0.jar
dbgpc.dbgp_jar = ../dbgp/target/dbgp-4.4.0.jar
dbgpc.vdmj_jvm = -Xmx1024m
```

You can use the DBGP Client 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 test.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 DBGP 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[1]. 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]>
[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]>
```

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


```
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 DBGP, 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 DBGP 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 DBGP 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]>
```