1. vdmj.dbgp

Class Summary	
DBGPReader	The main class of the DBGp protocol reader.
DBGPCommand	A parsed IDE command.
DBGPOption	A parsed IDE command option.
DBGPFeatures	The set of features supported/set by the debugger.
StderrRedirector	Class to control redirection of stderr
StdoutRedirector	Class to control redirection of stdout

Enum Summary	
DBGPBreakpointType	The possible DBGp breakpoint types.
DBGPCommandType	The possible DBGp IDE command types.
DBGPOptionType	The possible IDE command option flags.
DBGPErrorCode	The possible status response error codes.
DBGPReason	The possible status response reason codes.
DBGPContextType	The possible variable name context types.
DBGPRedirect	The possible I/O redirection options.
DBGPStatus	The possible status responses.

Exception Summary	
DBGPException	A general purpose debugger exception.

The vdmj.dbgp package contains classes that implement the DBGp remote debugging protocol defined in [6]. This allows VDMJ to load a set of specifications and evaluate/debug them under the remote control of another process – usually a GUI IDE, such as Eclipse.

As described in [6], the principle behind DBGp is that the IDE launches the debugged process (ie. VDMJ in our case), and that process connects back to the IDE using a TCP/IP connection to a host/port specified by the IDE. The IDE then sends commands to the debugger on the connection, and the debugger acts on those commands, sending status and results back to the IDE via the same connection.

Because DBGp starts VDMJ, the principal class that handles the connection, DBGPReader, defines a "main" method. This is separate from the main method defined in the VDMJ class (see section 2.1). The command line arguments are as follows:

```
Usage: -h <host> -p <port> -k <ide key> <-vdmpp|-vdmsl|-vdmrt>
       -e <expression> | -e64 <base64 expression>
       [-w] [-q] [-log <logfile URL>] [-c <charset>] [-r <release>]
       [-coverage <dir URL>] [-default64 <base64 name>]
       [-remote <class>] {<filename URLs>}
```

The host and port identify the connection that must be opened back to the IDE to receive commands. The ide key is a value that must be passed back to the IDE during the initial connection. The VDM dialect is a mandatory option. The expression, optionally base64 encoded for special charsets, is the main expression to be evaluated, and the list of filenames identify the specification itself. Each filename is in the form of a file URI ("file:/..."). There are also several optional settings. The -w and -q options suppress warnings and information messages respectively; the -log option sets the location of

the VDM-RT real time log output; the -c option sets the character set for the specification; the -r option is a VDM language release name (classic or vdm10); the -coverage option defines where detailed coverage information is written (for use by the GUI); the -default64 option sets the default module or class name in base64; and the -remote option identifies a class name that will be called rather than the main VDMJ processing loop (for remote control – see section 3.2).

Depending on the dialect, DBGPReader creates an instance of VDMPP, VDMRT or VDMSL (see section 2.1), and uses it to parse and type check the list of files passed. The DBGp protocol assumes the "program" being debugged is already compiled correctly, so if there are any syntax or type checking errors, these are sent to the VDMJ process' standard output and the process quits with an error exit code. The protocol has no mechanism for returning these errors to the IDE.

If the specification is clean (warnings are permitted), an Interpreter object encapsulating the parsed/checked specification is obtained from the VDMSL, VDMPP or VDMRT object, and this, along with the host, port, ide key and expression are passed to the constructor of a new DBGPReader object. The constructor saves the values, and sends the DBGp initialization message back to the IDE. Lastly, the main method calls the run method of the reader. When this method returns, the VDMJ process quits with a success error code (0). If any exceptions are thrown from run, it quits with an error exit code.

The DBGPReader run method is a read/execute loop, reading DBGp commands from the IDE connection, processing them, sending back any responses, and then looping. The return value from the private methods to handle each type of command indicate whether the run method should keep looping or return (for example, the "detach" command would indicate that the loop should return, though most commands keep looping).

All DBGp commands are in a simple textural format, similar to the format of a UNIX command (see [6]):

```
command -op1 v1 -op2 v2 ... -- data
```

This text is parsed by a method which produces a DBGPCommand object; parsing errors throw a DBGPException, which is caught and used to build an error status response before returning to the run loop.

Successfully parsed commands are passed to one of a number of processCmd methods which handle each type of command. The general form of these is to validate the options passed (held in the parsed DBGPCommand object), and send back an error response if not correct; and then act on the command, sending back a successful status, possibly combined with information being requested by the IDE.

Responses sent to the IDE are all XML formatted messages (see [6]). A collection of private methods in DBGPReader take the raw text response from a processed command in a StringBuilder, and use this to create an XML response message and send it on the connection to the IDE .

All of the DBGp commands which interact with the running specification do so by making method calls on the Interpreter object passed to the DBGPReader's constructor. Note that this is the abstract Interpreter class, not the concrete ClassInterpreter or ModuleInterpreter, so the DBGPReader will work with VDM-SL, VDM++ or VDM-RT.

When the specification is executed (via the DBGp "run" command), the interpreter's execute method is called, passing the expression to be evaluated. The DBGPReader instance is also passed to the execute method; this is stored in the thread context information for the main VDM thread, and allows breakpoints to find the connection to the IDE.

The interpreter's execute method does not return until the specification has been fully evaluated, so this raises the question of what happens at breakpoints, when information must be passed to the IDE and more instructions received.

Breakpoints normally use the DebuggerReader class to interact with the user on the command line before continuing execution (see section 2.13). But when the process is being debugged by an IDE, the breakpoint will find the DBGPReader object associated with its thread context via the DebugLink class, and call its "stopped" method. This sets the state of the connection to "break" and sends a status message to the IDE to let it know that execution has stopped at a breakpoint. It then enters the

"run" method to process IDE commands as it did originally (in fact this is a recursive call, since the original run call is on the stack, having called the execute method). Note that some DBGp commands are only acceptable when the connection is in the "break" state, such as "step_into" and "stack_get". Expressions can be evaluated in the "break" state and will be evaluated in the context of the breakpoint (ie. local variables are visible).

A "continue" command from the IDE will cause the recursive run call to return, and the flow of control will return to the breakpoint and from there back into the main execution. Further breaks may occur, but eventually, the main evaluation will complete and the original run method call will regain control. Note that the debugged process does not automatically quit at this point. It enters a "stopped" state, where further (restricted) IDE commands are possible, such as to retrieve the final evaluation result.

The DBGp protocol allows multi-threaded programs to be debugged. In this case, each thread opens its own connection to the IDE (on the same host/port), though the opening of the connection is delayed until the thread has something to say – perhaps at a breakpoint – to avoid bombarding the GUI with thread create/death connections that do nothing. The IDE is responsible for sending commands for each thread on the appropriate connection. As far as the debugged process is concerned, these connections are completely separate. Therefore, when VDM starts a new thread, the creating thread's DBGPReader is "cloned" to create a new object which will lazily open its own connection to the IDE, and be stored in the new thread's context. When the second thread reaches a breakpoint, it will respond to the IDE on its own connection.

The IDE can request that stdout/stderr output from the debugged process be sent to the IDE rather than (or as well as) to the usual console. If such a command is received, the Console class is called (see section 2.16), and a Redirector is added to the appropriate stream. These objects are passed a DBGPReader reference and use this to send output to the IDE when directed to do so.

Note that the main DBGPReader implements the common VDMJ DebugLink interface.

1.1. Comments

It would probably be better to use some sort of XML handling package to build the IDE responses, rather than hand-crafting them. Though the XML involved is very simple.

Asynchronous debugging could be done if checkpoints (ie. at the start of every expression or statement) check the status of the DBGp connection. That is simple to arrange, as the object is in the thread's context, but the overhead may be large (calling the input stream's "available" method). It might be acceptable to (say) only test the stream every 100 or 1000 operations, at the risk of making the threads unresponsive to asynchronous breaks.

A better solution would be to have asynchronous listening threads that call the signal method of SchedulableThreads, though that would double the number of threads in the system.

Note that the Console class only has one Redirector wrapped around an output stream. That means that all output is redirected to one particular DBGp connection, rather than the output from different threads being directed to each thread's DBGp connection. The thread that receives all the output is the one that last sent the IDE command to redirect it.

Currently, the values of all variables are sent back to the IDE as strings. To enable the structure of aggregate types to be expanded by the IDE, we need to define an XML Schema definition to describe them. This has not yet been done. When it is defined, it should be possible to change the Type class hierarchy to produce XML Schema to describe themselves, and to change the Value class hierarchy to "print" themselves in that schema.